Institutt for energiteknikk
OECD Halden Reactor Project

The Software Bus

Communication System

USER'S GUIDE

SOFTWARE BUS
DOCUMENTATION

Revision: 4.12

Institutt for energiteknikk
OECD Halden Reactor Project

Published by : Institutt for Energiteknikk, OECD Halden Reactor Project
Date : March 2017
Revision : 4.12

SOFTWARE BUS DOCUMENTATION

Table of Contents

1 Introduction

This chapter introduces the Software Bus, explains how to use this manual, describes other related documentation, and tells you how to get support and join the Software Bus Interest Group.

1.1 The Software Bus

The Software Bus (SWBus) is an object-oriented communications system that manages a dynamic set of distributed objects. Applications that use SWBus objects are able to share data and functionality with other processes running on different systems across a network.

The SWBus uses a TCP/IP-based message system that is completely hidden from the application programmer. Conversion of data transferred between different machine architectures is handled transparently, removing the enormous burden of writing and maintaining reliable code using conventional low-level communication tools. By using high-level object-oriented concepts, the SWBus enables the easy development of powerful distributed software systems.

The SWBus consists of a C library, that is linked into SWBus processes, and a port-mapper, that manages host-port addresses of such processes. A high-level Application Programmer's Interface (API) provides programmers with access to the functions in the library.

1.2 Using this manual

This manual describes how to develop distributed applications using the SWBus. It assumes that you are familiar with C or C++. An understanding of object-oriented concepts is an advantage, but not a prerequisite.

The manual is divided into five chapters and three appendices. This chapter describes this manual, conventions used, related documents, and how to get support and join the interest group.

Chapter 2 'Software Bus Concepts', introduces the key features of the SW-Bus and the concept of distributed object computing.

Chapter 3 'Getting Started', provides an overview of the file structure of the installation, and explains how to set up a suitable programming environment and how to run processes that rely on the SWBus for distributed object services.

Chapter 4 'Basic Programming', explains how to set about writing distributed object computing software using the SWBus library. It introduces all of the basic library functions in a tutorial-like manner, providing a solid basis for exploring Chapter 5.

Chapter 5 'Advanced Programming', looks at a number of advanced programming techniques that can be used when writing applications using the SWBus library.

Appendix A 'Software Bus API Quick Reference', summarises the functions in the SWBus library.

Appendix B 'Basic Example', contains the source code for the example used throughout Chapter 4.

Appendix C 'Advanced Example', describes the example application used in Chapter 5. The complete source code is provided together with instructions on how to compile and run the code.

1.3 Typographic conventions

This manual uses the following conventions:

- BUSPATH refers to the directory where the SWBus distribution is installed. This is typically */usr/local/swbus*.
- The bold Courier font is used to indicate source code. For example: **`SbInit("Server", NULL, NULL);`**
- Function names in the main text are indicated by emphasised, italic, text. For example: ***SbInit***.
- Function arguments in the main text are indicated by italic text. For example: *source*.
- Constants, such as class names, and variables in the main text are indicated by emphasised text. For example: **~periodic**.

1.4 Related documents

Software Bus Reference Manual

The Reference Manual contains detailed descriptions of the purpose and usage of the SWBus API functions, as well as other reference information relevant to using the SWBus.

On-line Documentation

The on-line UNIX 'man' pages provide the information in the Reference Manual in an easy-to-access on-line form.

1.5 Support

We hope that you enjoy using the SWBus and that it helps you to develop reliable, powerful, distributed computer systems faster and more coveniently than using conventional low-level communication tools.

If you have questions or comments about this software, then please contact us by e-mail at the following address:

softbus@hrp.no

1.6 The Software Bus Interest Group

The Software Bus Interest Group is a forum for general discussion about the Software Bus. Send a request by e-mail to our support address (**softbus@hrp.no**) to join.

All e-mail sent to the Software Bus Interest Group is distributed to everyone on the mailing list. The e-mail address is:

softbus-ig@hrp.no

Everyone that develops software using the Software Bus is encouraged to join the interest group, to keep up-to-date with future plans and new releases, and to discuss possible new features, ideas and general usage.

1.7 Web site

WWW pages for the SWBus are available at the URL:

http://www.ife.no/swbus/

2 Concepts

This chapter provides a general overview of the Software Bus and the concepts related to working with distributed objects.

2.1 SWBus overview

The SWBus consists of a C library and a port-mapper.

The C library **swbus** is the main part of the SWBus and is linked into all processes that use it's services. The library manages SW-Bus objects, marshalling[1], and low-level communication between processes. It also manages program flow, so that a SWBus process can react to incoming data, and it maintains a periodic handler mechanism that enables tasks to be performed at regular intervals, even when no data is being received from remote processes or via other input channels.

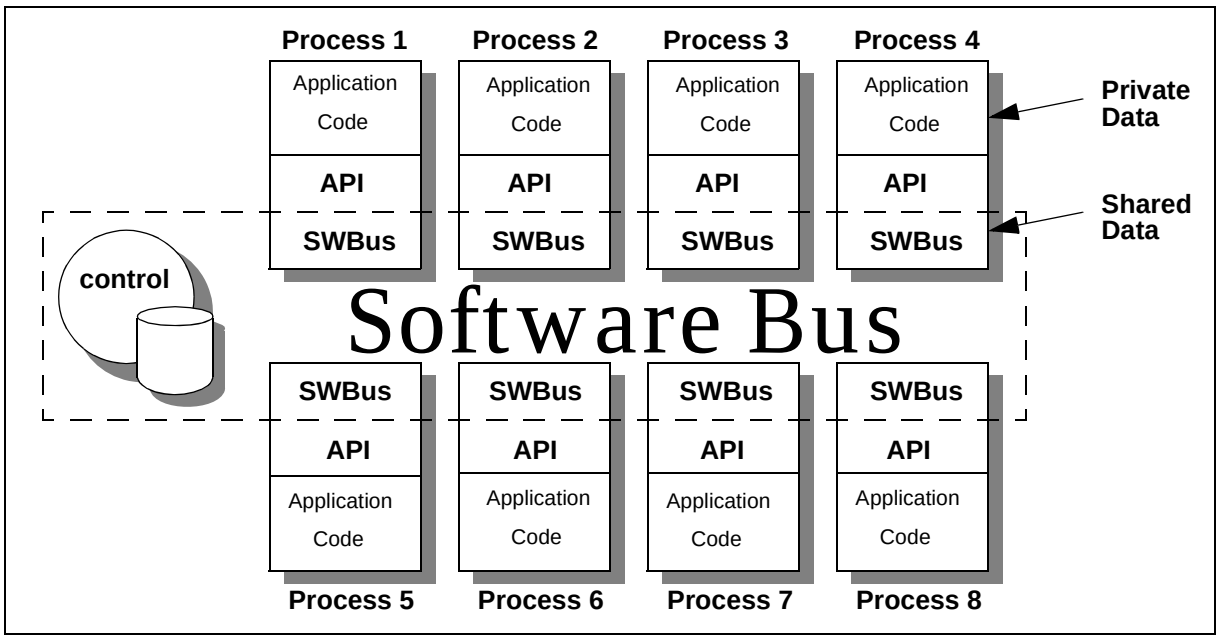
The port-mapper is called **control**, and runs on any computer on a network. It registers host and port addresses using process names supplied by the processes themselves when they initialise, and it returns host and port addresses for given process names each time a process opens a connection to another SWBus process. The port-mapper's database is global. SWBus processes that communicate with each other use a single port-mapper.

1 Marshalling is the process of reorganising data before and/or after it is sent to processes running on different hardware platforms so that it can be packed/unpacked correctly. It is needed because different hardware platforms typically store data in different formats (size, order, and alignment).

The port-mapper runs in the background like a daemon and is normally left running, even when no SWBus processes are running. It is small and uses practically no system resources.

Data and functionality, in a process, that is managed by the SWBus is available to other processes, both on the local host and on other machines in the network. The result of this is an integrated collection of processes that share data and functionality.

A typical running system looks like this:



Each process is a component of a distributed object computing system. Typical components might be a display component, a trend-logging component, a simulator component, a printer management component, and so on. Application-independent components, such as trend-loggers, can be designed to be reusable. A well-designed trend-logger can log the value of a SWBus object in any process at regular intervals, and can therefore be used in any application where a trend-logger is required.

To write programs that use the SWBus library, an Application Programmer's Interface (API) is used. It enables SWBus functions to be called from any programming language that permits the calling of C functions. It provides a small number of general functions that provide access to all SWBus objects.

New functionality can be added by overloading existing object methods or by adding new methods to objects.

Objects can be grouped together and treated as a single object. Invoking a method in a group object results in the method being invoked in every object in the group.

2.2 Distributed object computing

Distributed applications have traditionally been developed using relatively low-level tools. They are difficult to write and maintain, and often lack flexibility and extensibility. Using the SWBus, however, a higher level of abstraction is used to enable the easy implementation of distributed applications.

Features of object-oriented programming languages are used to separate the interface from the implementation and to provide powerful ways of accessing and storing data and functionality transparently between processes.

A well-designed distributed computing application is

- **efficient** - processing can be spread and done in parallel.
- **extensible** - new components can easily be integrated and the system be reconfigured dynamically.
- **reliable** - objects can be replicated so that network disruption can be coped with gracefully.
- **cost-effective** - distributed applications are relatively easy to implement and maintain. Also computing resources can be shared.

2.3 Objects

An important concept in the SWBus is the object. Everything in the SWBus is a uniquely identifiable object. The concepts from object-oriented programming that are used in the SWBus include

- **encapsulation** - SWBus objects are typically collections of attributes and methods. These can be accessed using a standard interface from outside the object. Implements information hiding and abstraction.
- **polymorphism** - the same message sent to different objects results in behaviour depending on the object that receives the message.
- **inheritance** - new classes and behaviour are defined based on existing classes. A new class is created by creating a sub-class of an existing class, thus inheriting the parent class' members.
- **dynamic binding** - objects can be created in a process by remote processes. Therefore bindings between objects, attributes, methods and their names can be modified while a program is executing.

As stated above, everything in the SWBus is an object; classes are instances of a 'class' class, functions are instances of a 'function' class, processes are instances of a 'process' class, and so on.

2.4 Classes

Classes are in some ways similar to C structures and are used as templates for creating objects. Attributes and methods can be added to classes to give them the state and behaviour desired.

Classes in the SWBus are dynamic, so when a class definition is modified, all instance of the class are also modified. The ability to add and remove members of a class at run-time is a very powerful mechanism.

A number of pre-defined classes exist that can be easily extended, including

- an ’empty’ class
- an ‘array’ class
- a ‘process’ class

A class can be extended by creating a sub-class of it and providing it with additional attributes and/or methods. The sub-class inherits all of the parent class’ properties, and thus can be used to extend it.

A number of simple data types are also provided. Simple data types cannot be created at run-time. They include different kinds of integers, floats, and characters.

2.5 Functions and methods

A function declaration contains information about the input and output parameters of a function, as well as a pointer to where the actual code that is to be executed resides. The input and output parameters are SWBus objects.

Once a function has been declared, and an instance of the function has been made, it can be called by any SWBus process. It is uniquely identifiable, just like other SWBus objects.

Functions that are members of classes are called methods. They are added to classes in the same way as attributes. Methods must have unique names within the scope of an object, but different classes can have methods with the same name. Thus the same call operation can result in different methods being invoked, depending on the context.

2.6 Immediate and proxy objects

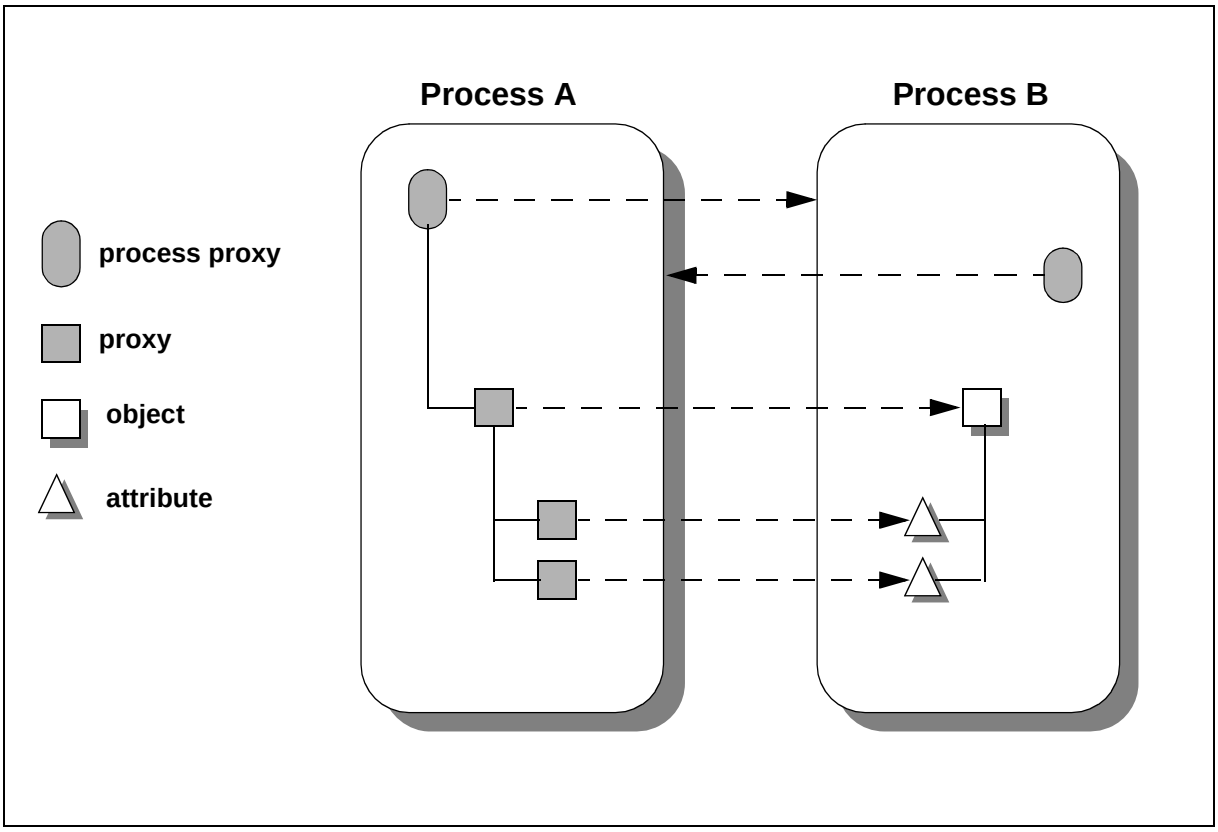
Two important concepts in the SWBus are those of ‘immediate’ and ‘proxy’ objects.

An immediate object is an object that is located in a local process, typically with a data representation in the address space of the process. Immediate objects are accessed directly by the local process.

A proxy object, on the other hand, is a local representation of a remote object. Proxies are sometimes called global references. Seen from the local process, a proxy looks like an immediate object, but when a proxy object is accessed or modified, the SWBus transparently accesses the actual, remote, object. A method invoked in a proxy object is, therefore, actually invoked in the process in which the object it represents resides.

When a process is connected to a remote process, the remote process is represented in the local process as a proxy. The objects in a remote process are accessed by the local process as elements in the remote process proxy.

A simple illustration of proxies is shown below:



Process B has a proxy to Process A and a local object with two attributes. Process A has proxies to Process B, to the object in Process B and to the attributes of the object.

Structured objects, such as arrays, groups and lists can contain elements that are proxies to remote objects, and not just immediate objects.

2.7 Links

The SWBus offers the possibility to link otherwise unrelated objects together. The only prerequisite is that the objects must either be instances of the same class or be instances of classes with identical data layouts.

When a source object is updated, the object's contents are copied to the target object, which may or may not be in the same process. When two objects in different processes are linked together, the source object is linked to a remote proxy object. Links are typically used to broadcast object values between and within processes.

Links are one-way. If a two-way link is required, two one-way links can be created.

In situations where two processes share objects of the same class, and there is a possibility that the class layout may change, the classes themselves can be linked, resulting in all class instances in both processes being modified if the source class is modified.

2.8 Main loop

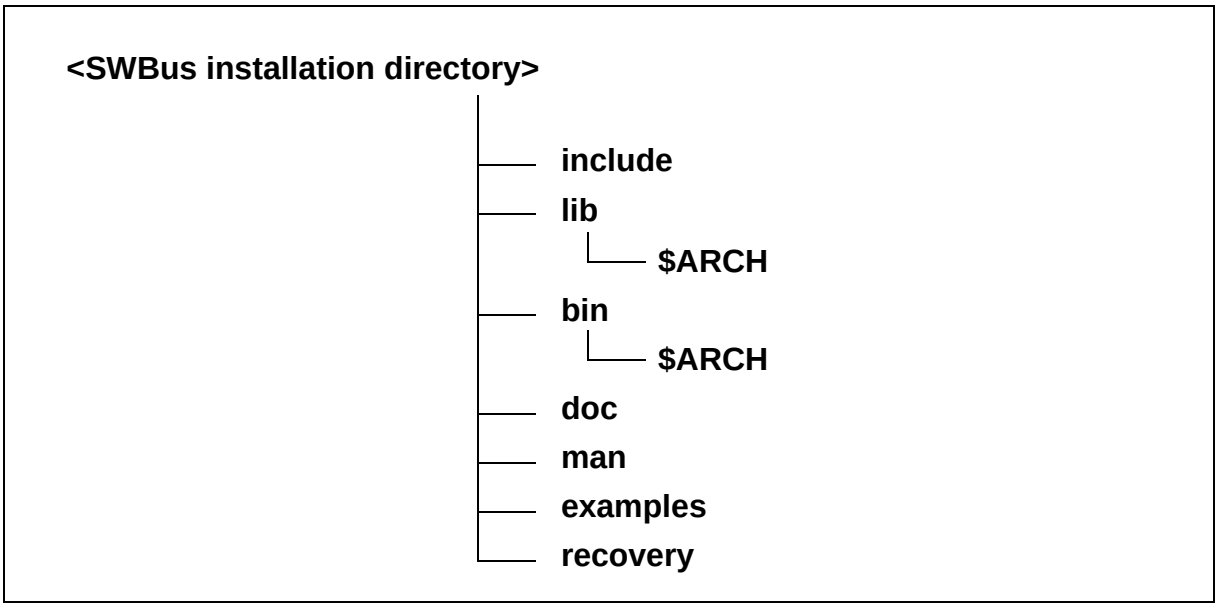
When the main SWBus objects required by a process have been initialised, control is usually passed to the SWBus' main loop. The main loop takes care of the handling of all input/output, calling of local SWBus functions by remote processes, and triggering of periodic handlers. The SWBus may be customized to replace the default main loop function by a user-defined one.

3 Getting Started

This chapter contains an overview of the directory structure of the SWBus installation directory, describes how to set up the SWBus environment, presents some guidelines for compiling SWBus applications, and describes the minimal SWBus application.

3.1 Directory structure

The SWBus will typically be installed in a directory easily accessible to several users, e.g. in */usr/local/swbus*. The environment variable **BUSPATH** should refer to the SWBus installation directory.



The *include* directory contains the SWBus header files with declarations of data types, constants, variables and functions.

The *lib* directory contains subdirectories for each platform supported by the SWBus. The SWBus library files are located in these subdirectories.

The *bin* directory contains subdirectories for each platform supported, and in these subdirectories the executable programs **control** and **controlutil** can be found.

The *doc* directory contains postscript files for the SWBus documentation and a text file called *ReleaseNotes* describing the latest SWBus news.

The *man* directory contains Unix manual pages for the SWBus. Include *$BUSPATH/man* in your MANPATH environment variable to enable the man program to locate the SWBus manual pages.

The *examples* directory contains source code for the tutorials in this document, and a performance test for the SWBus. A makefile is included as well.

The *recovery* directory is where control stores its recovery file, and is initially an empty directory. The recovery file enables control to regain its previous state in case of a crash or shutdown.

3.2 SWBus environment

The SWBus requires a couple of extensions to your environment: the services database must be extended and some environment variables are mandatory. Please read this section carefully and modify your environment as described before trying to compile and run any SWBus application.

The services database

The SWBus requires three entries in the services database of your system: one entry for **control**, and two entries defining a range available for SWBus applications. The entry required by **control** is called "softbus_name". The other two entries are called "comix_minfreeport" and "comix_maxfreeport". The range between these two should be large enough to cover at least twice the maximum number of connections you simultaneously want to put on your SWBus. Note that the port numbers within this range must all be free, no other programs sould be allowed to use these ports.

Here is an example of what the entries may look like:

```
softbus_name        17000/tcp
comix_minfreeport   17100/tcp
comix_maxfreeport   17499/tcp
```

The services database is a plain text file. On Unix platforms, it is located at */etc/services*, and on Windows platforms, it is normally located at *%windir%\system32\drivers\etc\services*.

On Unix systems, information on the services database can be found with the "man services" command. Normally, super-user privileges are required to update the services data base. Contact your local system administrator for help.

Note also that if your system is set up to use NIS for mapping of the services database, you may need to update your NIS server. On some Unix systems, access to the services database is controlled with the file /etc/nsswitch.conf which determines whether to use a local file or use NIS mapping. See "man nsswitch" for details.

Environment variables

A number of environment variables are used by the SWBus. Some are mandatory, some are optional. They are all listed below. Note that the SWBus documentation uses Unix syntax when referring to environment variables. For instance, we refer to the location of the library files as $BUSPATH/lib/$ARCH, while on Windows, the syntax would be %BUSPATH%\lib\%ARCH%.

BUSPATH
: The name of the SWBus installation directory, e.g. */usr/local/swbus*. It is used by **control** to locate the *recovery* directory and by the SWBus to start **control** automatically, if **control** is not running and it was supposed to be running on the local host. Setting the BUSPATH environment variable is mandatory.

ARCH
: The architecture of the computer. One of the following values should be used: *hpArch, hpi32Arch, hpi64Arch, linuxX86Arch, linuxX86_64Arch, macX86Arch, winArch, win64Arch*. Setting the ARCH environment variable is mandatory.

LIB
: Search path for libraries on *winArch*. LIB should include $BUSPATH/lib/$ARCH and the path to ws2_32.lib. Setting the LIB environment variable is not necessary on Unix platforms but mandatory on Windows platforms.

CONTROLHOST
: Specifies the host on which **control**, the port-mapper, is assumed to be running. If CONTROLHOST is not set, localhost is assumed.

CONTROLPORT
: Specifies the port **control** will use when listening for clients. If CONTROLPORT is not set, a default port will be used. The spesified port number may be an integer value, an offset (an integer preceeded by + or -) or a port name.

BUSTIMEOUT
: The maximum duration (in milliseconds) a SWBus process should wait for an answer from a remote function before considering it lost. Default value -1, infinite.

BUSRETRY — The duration (in milliseconds) between each attempt by a SWBus process to connect to another process. Default value is 4.000.

BUSCONNECT — The maximum duration (in milliseconds) before the application should give up trying to connect to another SWBus process. The default value is -1, i.e the process will try forever at intervals specified by BUSRETRY.

BUSPINGTIMEOUT — The maximum duration (in milliseconds) control should accept a missing ping signal from the process before issuing a process-is-dead message to all other processes. The value of BUSPINGTIMOUT is read by each process at startup and transferred to control when the process registers. Default value -1, indicating that the ping-feature is disabled.

BUSPINGINTERVAL — The interval (in milliseconds) between each ping message (I-am-alive) sent by a processes to control. If not set, the process will issue ping messages at an interval of half the value of BUSPINGTIMEOUT.

BUSCONNECTTIMEOUTS — This variable should consist of three white-space-separated integers. The first integer specifies the time to wait for a connect-request towards another SWBus process to complete. The second integer specifies the time to wait for a connect-request towards control to complete. The third integer specifies the number of connect-requests to send to control before concluding that control is not available. Default value for this variable is 250 500 5

BUSHOSTFILE — The full path of a text file containing a set of host names. Control will read the file at start-up and deny access for any process running on a host not included in the file. If the environment variable is not set, control will accept processes running on any host.

BUSTCPDELAY — The SWBus uses TCP_NODELAY to remove delays when sending data on a socket. If this behaviour is undesirable, BUSTCPDELAY can be set to a nonzero value to avoid using TCP_NODELAY.

MANPATH — An environnment variable used by the Unix man program. Include $BUSPATH / man in your MANPATH environment variable to enable the man program to display the manual pages from the SWBus Reference Manual.

3.3 Compiling and linking SWBus applications

How to compile and link a SWBus application is described below. There are separate descriptions for Unix and Windows platforms.

In the subdirectory *examples* of the SWBus installation directory, makefiles for the example source code can be found. These makefiles can be used as a basis when constructing your own makefiles for your SWBus applications.

Compiling and linking on Unix platforms

1) The C compiler must be told where to locate the SWBus header files. Use the compiler option **-I$BUSPATH/include**.
2) Your architecture must be defined when compiling. Use the compiler option **-D$ARCH**.
3) The linker must be told where to locate the SWBus library file. Use the linker option ***-L$BUSPATH/lib/$ARCH***.
4) The SWBus library must be included when linking your application. Use the linker option **-lswbus_debug** or **-lswbus**.
5) Some platforms may require additional libraries to be included when linking.

To summarize, typical compile and link commands on the ***linuxX86Arch*** platform, would be:

```
gcc -c -D$ARCH -I$BUSPATH/include hello.c
gcc -L$BUSPATH/lib/$ARCH hello.o -lswbus -lm -lpthread -lrt -ldl
```

Compiling and linking on Windows platforms

1) The C compiler must be told where to locate the SWBus header files. Use the compiler option **/I%BUSPATH%\include**.
2) Your architecture must be defined when compiling. Use the compiler option **/D%ARCH%**.
3) If your application uses the fd_set structure and SWBus functions for manipulating suchvariables, use the compiler option **/DFD_SETSIZE=1024**.
4) To control alignment of structs, the **/Zp8** option (default) must be used.
5) There are several files for the SWBus library on Windows, select the one that suits your application according to the table below. The compiler option /MT specifies multithreading, and /MD specifies multithreading with DLL version of the run-time routines.

Compiler option	SWBus library	C library
/MT	swbusMT.lib	LIBCMT.LIB
/MTd	swbusMTd.lib	LIBCMTD.LIB
/MD	swbusMD.lib	MSVCRT.LIB
/MDd	swbusMDd.lib	MSVCRTD.LIB

6) The linker must be told where to locate the SWBus library file and the Windows socket library. The environment variable LIB must include ***%BUSPATH%\lib\%ARCH%*** and the path to ws2_32.lib.

7) The desired SWBus library and the library ws2_32.lib must be included when linking your application.

To summarize, typical compile and link commands would be:

```
cl /c /D%ARCH% /I%BUSPATH%\include hello.c
cl hello.o swbusMD.lib ws2_32.lib
```

3.4 Running a SWBus application

Before running a SWBus application, ensure that the portmapper, **control**, is running on a host in your network, and that the environment variable CONTROLHOST specifies the name of that host.

control should be started before any other SWBus process. It should normally be started in the background like a daemon process and be left running, even while no SWBus processes are running. It is small and uses practically no system resources.

When started, **control** will respond with something like: "Control (SWBus Release 4.0 June 2003)".

To check the state of **control**, the utility program **controlutil** can be used. Try "controlutil -h <yourControlHost> -q". Please refer to the manual page for controlutil for more details.

3.5 The minimal SWBus application

The smallest possible SWBus application is outlined below.

MINIMAL APPLICATION

```
#include <swbus.h>

int main( int argc, char* argv[] )
{
   SbInit(argv[0],NULL,NULL);
   SbLoop();
   SbExit();
   return 0;
}
```

First the SWBus header file *swbus.h* is included to make SWBus data types, constants, variables and functions available to the compiler. Then the SWBus initialization function, ***SbInit***, is called. ***SbInit*** initializes the SWBus and registers the process with the port-mapper, **control**. After initializing the SWBus, this program enters the SWBus main loop, waiting for other SWBus processes to perform some actions.

4 Basic Programming

This chapter contains a thorough description of a basic SWBus application. It serves as an introduction to writing SWBus programs. Its main focus is on demonstrating basic SWBus functionality rather than on optimal or fault tolerant programming.

4.1 Name conventions

All functions in the SWBus API have the prefix Sb (Software Bus), e.g. ***SbCreate***, ***SbAdd***, ***SbPut***.

C data-types used in the SWBus API have the prefix SbT (Software Bus Type), e.g. **SbTSTI**, **SbTCode**, **SbTPeriodicParameter**.

Constants have the prefix SbC (Software Bus Constant), e.g. **SbCNull**, **SbCError**, **SbCWait**.

Predefined SWBus classes use the prefix SbCC (Software Bus Constant Class), e.g. **SbCCInteger**, **SbCCEmpty**, **SbCCPeriodic**.

4.2 The basic SWBus data type

The basic SWBus data type is **SbTSTI**. A variable of type **SbTSTI** can hold an id representing a SWBus object. The id is referred to as the Symbol Table Index (STI) of the object. Everything in the SWBus is represented by its STI: constants like **SbCNull**, **SbCError** and **SbCOK**, basic SWBus classes such as **SbCCInteger**, **SbCCString**, and **SbCCEmpty**, processes such as **SbCPLocal**, and user defined variables, classes, and functions.

Most functions in the SWBus API take an STI (a variable of type **SbTSTI**), as their first input argument, and most also return an STI.

4.3 About the example

The example used throughout this chapter is an application consisting of two programs called *simpleServer* and *simpleClient*. The server demonstrates how to create and use basic SWBus objects such as variables, functions, classes, instances, and arrays before entering a main loop and waiting for the client to perform some actions.

The client initiates a connection to the server, demonstrating how to use variables and functions from the server, and then closes the connection.

The complete example can be found as C source code in the *examples* subdirectory of the SWBus installation directory. Compile it using the *makefile* provided, and run it to see how it works. Guidelines for how to compile and run the example programs are provided in the *README* file of the *examples* directory.

We begin by describing the server code in detail. The client code is smaller and will be described in Sections 4.18 through 4.22, page 33 to page 37.

4.4 Initializing the SWBus

Prior to using any other SWBus API functions, the function ***SbInit*** must be called. ***SbInit*** is responsible for initializing the program as a SWBus process and registering it with the portmapper, **control**.

SIMPLESERVER

```
#include <swbus.h>

int main()
{
   char* serverName = "simpleServer";
   if ( SbInit( serverName, NULL, NULL ) != SbCOK )
   {
      printf( "SbInit failed for process %s\n", serverName );
      exit( 1 );
   }
```

SbInit takes three character string arguments. The first is the name of the process, in this case "*simpleServer*". The second argument is the host where **control** is running. **NULL** is used to indicate that the value of the environment variable CONTROLHOST

should be used. If the environment variable is not set, ***SbInit*** assumes localhost. The third argument is reserved for future use and is currently ignored.

SbInit returns the constant **SbCOK** if the operation succeeds, otherwise **SbCError** is returned.

4.5 Creating an object of a basic type

When the Software Bus has been initialized, objects can be created. Here is how to create a local integer object:

SIMPLESERVER (continued)

```
SbTSTI myInt;
myInt = SbCreate( SbCPLocal, SbCCInteger, "myInt", SbCNull, NULL );
```

The function for creating SWBus objects is ***SbCreate***. It takes five arguments. The first argument is the STI of the process in which the object should be located. **SbCPLocal** is a predefined STI for the local process. The second argument is the STI of the class of the object; The class of integers is **SbCCInteger**. The third argument is the name of the object. Here, the object is called "myInt". Anonymous objects are allowed; Use **NULL** or an empty string to indicate an anonymous object. The fourth argument is the parameter to the constructor function of the object. Integers do not have constructors, so **SbCNull** is used. Constructor functions will be explained in detail later in this chapter. The last argument is a pointer to where the actual data is located. **NULL** indicates that the SWBus should allocate memory for the object.

SbCreate returns the STI of the new object if successful, otherwise **SbCError** is returned.

Naturally, there are other basic SWBus classes than **SbCCInteger**. There are classes for all standard C data types, several specialized SWBus classes, and SWBus programmers can also create their own classes. User-defined classes and arrays are described in Section 4.10 and 4.14. The list of predefined basic SWBus classes includes:

- **SbCCChar**
- **SbCCInt8** (8-bits integers)
- **SbCCInt16** (16-bits integers)
- **SbCCInt32** (32-bits integers). **SbCCInteger** is an alias for **SbCCInt32**.
- **SbCCFloat32** (32-bits floating point numbers). **SbCCFloat** is an alias for **SbCCFloat32**.

- **SbCCFloat64** (64-bits floating point numbers). **SbCCDouble** is an alias for **SbCCFloat64**.
- **SbCCString** (character pointers)
- **SbCCAddr** (other pointers)
- **SbCCLRef** (references to another object)
- **SbCCBuffer** (byte buffers)
- and several others.

Please refer to the Software Bus Reference Manual for details.

4.6 Assigning a value to an object

Now that we have created an integer object, let's store a value in it.

SIMPLESERVER (continued)

```
int i;
i = 5;
SbPut( myInt, &i );
printf( "Value %d assigned to object %s\n", i, SbName( myInt ) );
```

The function for assigning a value to an object is ***SbPut***. Its first argument is the STI of the object, the second is a pointer to where the value is located. ***SbPut*** copies the value at the specified address into the memory location of the object. The number of bytes to copy is determined by the class of the object, so ensure that your pointer points to a value of the same type as the class of the object.

The function ***SbName*** used in the ***printf*** statement above returns the name of the object, so the text "Value 5 assigned to object myInt" should appear on the screen after this piece of code has been executed.

4.7 Retrieving a value from an object

Naturally, we must be able to retrieve the value from our object.

SIMPLESERVER (continued)

```
int j;
SbGet( myInt, &j );
printf( "Value %d retrieved from object %s\n", j, SbName(myInt) );
```

SbGet is very similar to ***SbPut***. It takes the STI of an object and copies the value into a location pointed to by the second argument. The number of bytes to copy is determined by the class of the object.

The text "Value 5 retrieved from object myInt" should now be written on the screen.

4.8 Creating a function

Creating a function is done in three steps:

1) Provide the actual function code.
2) Declare a function class for the code.
3) Create an instance of the function class.

Step 1: Providing the function code

All user-supplied SWBus functions must conform to the SWBus function signature requirements. They must take four STI arguments, and return an STI. When the function is called by the SWBus, the four arguments are:

1) The STI of the owner of the function
2) The STI of the function itself
3) The STI of the input parameter
4) The STI of the output parameter

The function should normally return its output parameter, or **SbCError** if the function fails.

The classes of the input and output parameters (the third and fourth arguments of the function) are important. They are set up when the function class is declared in step 2 and determine the logical signature of the function. In the example below, they will both have the classes **SbCCInteger**, so the logical signature of the function is **`int f(int)`**. Thus, *in* and *out* will be STIs of integer objects.

SIMPLESERVER

```
SbTSTI theFunction( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   int k;

   SbGet( in, &k );
   printf( "Function %s.%s called with argument %d\n",
           SbName(o), SbName(f), k );
   k = 100 * k;
   SbPut( out, &k );
   return out;
}
```

The function retrieves the input value from its input parameter, multiplies it by 100, assigns the updated value to its output parameter, and return the output parameter to indicate successful completion of the operation.

This function will, in step 3, be set up to be a global function. So, argument *o* will be the STI of the process to which it belongs. Argument *f* is the STI of the function itself. In step 3, the function will be given the name "myFunc", so the text written to the screen when this function is called will be "Function simpleServer.myFunc called with argument x".

Step 2: Declaring the function class

SbDeclare creates a function class, and any instance of such a class will be a callable function.

SIMPLESERVER (continued)

```
SbTSTI funcClass;
funcClass = SbDeclare( NULL, SbCCInteger, SbCCInteger,
                       theFunction, SbCLC );
```

SbDeclare takes 5 arguments. The first is the name of the function class. In this example we use **NULL** to declare an anonymous class, as we are not interested in what it is called. The second and third arguments are the classes of the input and output parameters, respectively. Our function is, as we have seen, designed to take integers for both input and output, so we use **SbCCInteger** for both. The fourth argument is the function code, and the last argument is a constant that indicates that the function is compiled C code.

SbDeclare returns the STI of the function class, or **SbCError** if unsuccessful.

The standard signature of SWBus functions might mislead users to think that SWBus functions can only take a single input parameter and return a single output value. That is not true. Any SWBus class, including any user defined class, can be used for the second and third arguments of ***SbDeclare***. For example, to create a function taking to integers as input and returning a float and a double, create a SWBus class containing two integers for the input parameter, and another class containing a float and a double for the output parameter, and use these as arguments to ***SbDeclare***. Creation of classes is described in Section 4.10.

Step 3: Creating the function instance

The function instance is created with ***SbCreate***, just like the integer object created earlier.

SIMPLESERVER (continued)

```
SbTSTI myFunc;
myFunc = SbCreate( SbCPLocal, funcClass, "myFunc", SbCNull, NULL );
```

The function is created as an object in the local process, so it will be a global function. The class of the instance is the function class returned from ***SbDeclare***, and we choose to call the function "myFunc".

4.9 Calling a function

We will now see how to call the function that we have just created. Remember that the function was designed to take an integer input parameter and an integer output parameter.

SIMPLESERVER (continued)

```
i = 37;
SbPut( myInt, &i );
SbCall( myFunc, myInt, myInt );
SbGet( myInt, &j );
printf( "%s(%d) = %d\n", SbName(myFunc), i, j );
```

First we assign a value to the integer object *myInt* using ***SbPut***. Then we use ***SbCall*** to call our function, and finally we retrieve the result using ***SbGet*** and write the information to the screen.

SbCall takes three arguments: the STI of the function to be called, the actual input parameter and the actual output parameter. Here we use *myInt* both as input and output parameter.

The value returned by ***SbCall*** is the value returned by the function. As explained earlier this should be the output parameter if the operation was successful, and **SbCError** otherwise. In this case, the value returned by ***SbCall*** would be the STI of *myInt*.

The output on the screen after executing this piece of code will be "myFunc(37) = 3700".

4.10 Creating a SWBus class

When programming, one usually needs more complex data types than the basic SWBus classes. We would like to define our own classes, like we can define arbitrary complex structures in C. As with C structures, SWBus classes are constructed by adding attribute members of arbitrary types. In addition, like C++ classes, SWBus classes can contain methods.

Creating a subclass

A new SWBus class is always created by subclassing an existing class. The SWBus function to do this is ***SbSub***.

SIMPLESERVER (continued)

```
SbTSTI myClass;
myClass = SbSub( SbCCEmpty, "myClass", SbCNull );
```

SbSub takes an existing class and a character string and creates a subclass with the specified name. The predefined class **SbCCEmpty** is an empty class to be used as a starting point for creating user defined classes.

The last argument to ***SbSub*** is a parameter containing extra information about how to create the subclass. This is necessary when creating arrays, but for ordinary classes no additional information is required, so we simply use **SbCNull**. Creating arrays will be demonstrated in Section 4.14.

SbSub returns the STI of the new class, or **SbCError** if it fails.

The new class will inherit all members of its superclass, and will initially contain nothing more than its superclass. Now let's see how we can extend the subclass with attributes and methods.

Adding attributes

Attributes are added to a class using ***SbAdd***.

SIMPLESERVER (continued)

```
SbAdd( myClass, SbCCInteger, "theInt", NULL );
SbAdd( myClass, SbCCFloat,   "theFloat", NULL );
SbAdd( myClass, SbCCString,  "theString", NULL );
```

SbAdd takes four arguments. The first argument is the class which should be extended. The second is the class of the new attribute and the third is its name. The last argument is a pointer to a value which will be used as a default value for the new attribute if there are existing instances of the extended class when ***SbAdd*** is called.

SbAdd modifies the class so that the data layout of the class will always match the layout of a corresponding C struct.

SbAdd normally returns the STI of an attribute description object used by the class. **SbCError** is returned if ***SbAdd*** fails.

Adding methods

The methods of a class are created in very much the same way as global functions. First, function code must be provided, then a function class must be declared and added to the class using ***SbAdd***. When the class is instanciated, an instance of the function class will be available as one of the instance's methods.

Let's see how we can add a print-method to the class *myClass*. The method will retrieve values from the attributes of the object and output them to the screen. First the function code must be provided.

SIMPLESERVER

```
SbTSTI printObjectValues( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   int   theInt;
   float theFloat;
   char* theString;

   SbGet( SbId(o,"theInt"),    &theInt );
   SbGet( SbId(o,"theFloat"),  &theFloat );
   SbGet( SbId(o,"theString"), &theString );
   printf( "Object %s: theInt = %d, theFloat = %f, theString = %s\n",
           SbName(o), theInt, theFloat, theString );
   return out;
}
```

The signature of the method is the standard SWBus function signature. The object for which the method is invoked is supplied in the first argument when the method is called. So, *o* is the object that owns the print method that has been called. The second argument is the STI of the method itself, and the third and fourth arguments are the input and output parameters respectively.

The function ***SbId*** is used to retrieve an attribute of an object. It takes the STI of the object and the name of the attribute as input, and returns the attribute's STI. Using this value as input to ***SbGet***, we can copy the value of the attribute into a local variable in a function.

The print method does not use the input or output parameters, but it returns the STI of its output parameter to indicate successful execution.

Now that the function code is ready, we can declare the function class and add it to *myClass*.

SIMPLESERVER (continued)

```
   funcClass = SbDeclare( NULL, SbCNull, SbCNull,
                          printObjectValues, SbCLC );
   SbAdd( myClass, funcClass, "print", NULL );
```

The function class is declared using ***SbDeclare***, as described earlier. The print method does not use input or output parameters, so **SbCNull** is used for both. The function class is added to class *myClass* using ***SbAdd***, in the same way that the attributes were added earlier. Adding methods to a class does not affect the data area of the class' instances.

The print method has now been added to the class. Section 4.12 describes how to extract and call the method.

Adding a constructor

A constructor is a special method of a class that is called automatically when an instance of the class is created. Constructors are useful for initializing attributes and are frequently used in object-oriented systems.

A constructor is created exactly as any other method, but it must have the name "*~construct*". Let's see what the constructor for our class looks like.

SIMPLESERVER

```
struct myStruct
{
   int   theInt;
   float theFloat;
   char* theString;
};

SbTSTI constructor( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   struct myStruct* data = (struct myStruct*)SbData(o);

   if ( in == SbCNull )
   {  /* Default constructor */
      data->theInt = 1;
      data->theFloat = 1.1;
      data->theString = strdup( "default" );
      printf( "Default constructor of class %s called\n",
              SbName(SbClass(o)) );
   }
   else
   {  /* Constructor with integer input parameter */
      SbGet( in, &data->theInt );
      data->theFloat = 2.5 * data->theInt;
      data->theString = strdup( "some text" );
      printf( "Constructor of object %s called with value %d\n",
              SbName(o),data->theInt);
   }

   return out;
}
```

To optimize the constructor for speed, we use the function ***SbData*** instead of many calls to ***SbId***. ***SbData*** retrieves the data pointer of our object but is only useful when we know that the object is located in the local process. We explained earlier that when attributes are added to a class, the layout of the class exactly match-

es a corresponding C struct. We therefore know that the layout of an object of class *myClass* will match the layout of the struct *myStruct* because their attributes are organized in the same way. Thus, we can retrieve a pointer to the data area of the object by **`SbData(o)`** and assign values to the object using the retrieved pointer.

Note the introduction of a new SWBus function, ***SbClass***, in the first ***printf*** statement. It is a SWBus function that returns the STI of the class of an object.

Like any other method, a constructor can be set up to take input parameters. However, a constructor should be prepared to act as a default constructor when no input parameter is provided. If an object is part of a larger object, for instance an element in an array or an attribute of a more complex class, there is no way for the SWBus to supply an actual argument to the constructor, and in such cases **SbCNull** is used. Thus, the constructor should take special care if its input parameter is **SbCNull** and initialize attributes appropriately.

As in C++, output values from constructors are not returned to the caller and are therefore useless.

Now that the constructor code is ready, a function class must be declared and added to the class *myClass*, in the same way as for the print method.

SIMPLESERVER (continued)

```
funcClass = SbDeclare( NULL, SbCCInteger, SbCNull,
                       constructor, SbCLC );
SbAdd( myClass, funcClass, "~construct", NULL );
```

The constructor is declared to take an integer input parameter, but as we have seen, the function code is prepared for handling **SbCNull** as well. Note the name of the constructor method when added to the class; it must be "*~construct*".

Adding a destructor

A destructor is "the opposite of a constructor", a special method of a class that is called automatically when an instance of the class is deleted. Destructors are useful for cleaning up when an object dies.

A destructor is created in the same way as other methods, but must have the name "*~destruct*". Let's see what the destructor for our class looks like.

SIMPLESERVER

```
SbTSTI destructor( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   struct myStruct* data;

   printf( "Destructor of class %s called\n", SbName(SbClass(o)) );
   data = (struct myStruct*)SbData(o);
   free( data->theString );
   data->theString = NULL;
   return out;
}
```

The destructor of *myClass* uses ***SbData*** in the same way that the constructor does, and it frees memory allocated by ***strdup*** in the constructor.

Like other methods, a destructor can be set up to take input parameters. However, as for constructors, any destructor should be able to act as a default destructor without using the input parameter. If an object is part of a larger object, for instance an element in an array or an attribute of a more complex class, there is no way for the SWBus to supply an actual argument to the destructor, and in such cases **SbCNull** is used. Thus, the destructor should handle an input parameter with the value **SbCNull**.

As in C++, output values from destructors are not returned to the caller and are therefore useless.

Now that the destructor code is ready, a function class must be created and added to the class *myClass*, in the same way as for the print method. Note the name of the destructor when added to the class; it must be "*~destruct*".

SIMPLESERVER (continued)

```
funcClass = SbDeclare( NULL, SbCNull, SbCNull, destructor, SbCLC );
SbAdd( myClass, funcClass, "~destruct", NULL );
```

4.11 Creating an object of a user defined class

We will now see how to create an object of our class.

SIMPLESERVER (continued)

```
SbTSTI myObjectParam, myObject;
i = 2;
myObjectParam = SbCreate( SbCPLocal, SbCCInteger, NULL,
                          SbCNull, &i );
myObject      = SbCreate( SbCPLocal, myClass, "myObject",
                          myObjectParam,NULL);
```

To create the object, we need a parameter object for our constructor method. When we declared the constructor, the input parameter class was set up to be **SbCCInteger**, so we create an integer object for our parameter. Here we have used the address of the variable *i* as the last argument to ***SbCreate***, so the parameter object will have the same memory location as the variable *i*, and hence its value will be 2. The parameter object is created as an anonymous object.

The instance of *myClass* is then created as an object in the local process with class *myClass*. It is given the name "*myObject*" and the parameter to the constructor method is passed as an argument to ***SbCreate***.

4.12 Invoking a method of an object

To invoke a method of an object, the method itself must be extracted from the object using ***SbId***. The first time the method is extracted this way, an instance of the method is created. The method can then be invoked like an ordinary function, using ***SbCall***.

SIMPLESERVER (continued)

```
SbTSTI printMet;
printMet = SbId( myObject, "print" );
SbCall( printMet, SbCNull, SbCNull );
```

The print method of our class does not use input or output parameters, so **SbCNull** is used for both.

This piece of code outputs "Object myObject: theInt = 2, theFloat = 5.000000, theString = some text" to the screen.

4.13 Deleting an object

When we have finished using an object, it can be deleted using the SWBus function ***SbDelete***.

SIMPLESERVER (continued)

```
SbDelete( myObject, SbCNull );
SbDelete( myObjectParam, SbCNull );
```

SbDelete takes two arguments, the STI of the object to be deleted and a parameter to its destructor. Our destructor does not take arguments, so **SbCNull** is used.

The parameter object used for the constructor of *myObject* is deleted here as well, however, it could have been deleted immediately after *myObject* was created.

4.14 Creating an array

To create a SWBus array, we first need a suitable array class. An array class consists of a base class and a number of elements, so the class for arrays of five integers is different from the class for arrays of seven integers.

An array class is created using ***SbSub*** in the same way as any another class, but, as mentioned earlier, ***SbSub*** needs extra information to make an array subclass. The extra information required is the actual base class and number of elements. This information is provided to ***SbSub*** by the use of an extra parameter of class **SbCCArrayParameter**.

SIMPLESERVER (continued)

```
#define NumArrayElements 5
SbTArrayParameter paramData;
SbTSTI param, arrayClass, myArray;

paramData.baseClass = myClass;
paramData.count     = NumArrayElements;
param = SbCreate( SbCPLocal, SbCCArrayParameter, NULL,
                  SbCNull, &paramData );
arrayClass = SbSub( SbCCArray, NULL, param );
SbDelete( param, SbCNull );
myArray = SbCreate( SbCPLocal, arrayClass, "myArray",
                    SbCNull, NULL );
```

The class **SbCCArrayParameter** matches the C struct **SbTArrayParameter**, and has two attributes, *baseClass* and *count*. In our example, we use our previously created class *myClass* as the base class for the array, and set *count* to 5. So, we will create an array with five elements of class *myClass*.

A parameter is created using ***SbCreate*** with the address of the C struct as value pointer, just like we did for the constructor parameter for *myObject* in Section 4.11.

An array class is then created using ***SbSub***, with the *param* object providing the extra information. Array classes are always created by subclassing **SbCCArray**, a predefined superclass for all array classes. We use the **NULL** pointer as the name of the class, as we are not interested in what it's called. After the array class has been created, it's constructor parameter is no longer needed, so we delete it using ***SbDelete***.

Finally, an array instance is created using ***SbCreate***. It is created as an object in the local process, using our new array class as the class of the object. The array is given the name "*myArray*".

Some words about constructors and destructors

We did not create a constructor or destructor for our array class, so there is no obvious constructor to call when it is instanciated. However, the elements of the array are instances of the class *myClass*, and *myClass* has a constructor. Thus, the constructor of *myClass* is called for each element of the array. As stated when the constructor of *myClass* was created, the constructor will not be passed any input parameters when the objects are components of a larger structure, so the constructor will be called as a default constructor with **SbCNull** as input parameter.

On the other hand, if a constructor for the array class had been defined, then that constructor would have been called when the array was created, and the constructor of *myClass* would not have been called at all to prevent double initialization.

This holds for destructors as well. When no destructor is created for the array class, the destructor function of the base class will be called for each element when the array is deleted. If a destructor for the array class exists, that destructor will be called instead.

4.15 Accessing elements of an array

An element of an array is an object and it is extracted using ***SbIndex***.

SIMPLESERVER (continued)

```
for ( i = 0; i < NumArrayElements; i++ )
{
   element = SbIndex( myArray, i );
   SbCall( SbId( element, "print" ), SbCNull, SbCNull );
}

SbDelete( myArray, SbCNull );
```

SbIndex takes two arguments, the STI of the array and the index of the element, and returns the STI of the element. The elements of our array are instances of the class *myClass*, and the example above iterates over the array, and invokes the print method of each element.

When the complete example is run, the output from these method invocations will show that the elements were initialized by the default constructor of the class *myClass*. For example, for the element with index 3, the text "Object [3]: theInt = 1, theFloat = 1.100000, theString = default" will be output to the screen.

Finally the array is deleted, and the destructor for each element is called.

4.16 Entering the main loop

Up to now everything we have demonstrated simply creates different kinds of local objects and calls local functions. However, there is no advantage in using the SWBus unless the program communicates with other processes. Simplifying interprocess communication is what the SWBus is really all about.

For a process to be accessible from another process, it must enter its main loop. Entering the main loop enable a SWBus process to receive input from other processes.

Before we enter the main loop, let's declare a function to terminate it, otherwise the server will remain in the main loop forever. As usual, we begin by providing the function code.

SIMPLESERVER

```
SbTSTI terminate( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   printf( "Server asked to stop\n" );
   SbEndLoop();
   return out;
}
```

The function above calls ***SbEndLoop***, the SWBus API function to terminate the main loop. So, when someone calls this function, the main loop of the server will be terminated the next time control is returned to the main loop.

The function class must be declared and a function instance must be created in order to make it a callable SWBus function.

SIMPLESERVER (continued)

```
    SbTSTI termFunc;
    funcClass = SbDeclare( NULL, SbCNull, SbCNull, terminate, SbCLC );
    termFunc  = SbCreate( SbCPLocal, funcClass, "terminate",
                          SbCNull, NULL );
```

Now, we are ready to enter the main loop.

SIMPLESERVER (continued)

```
    printf( "%s entering main loop ...\n", SbName(SbCPLocal) );
    SbLoop();
```

SbLoop is the function to call to enter the main SWBus loop. It does not take any arguments, and the program will loop inside ***SbLoop*** until ***SbEndLoop*** is called.

4.17 Leaving the SWBus

When the main loop terminates, our server has finished its tasks, so it will leave the SWBus and exit.

SIMPLESERVER (continued)

```
    SbExit();
    return 0;
} /* End of main */
```

SbExit unregisters the process with **control**, and leaves the SW-Bus.

4.18 Initializing the client program

Now, let's have a look at the client, to demonstrate how to connect to remote processes and how to access remote objects and call remote functions.

First of all, the client program has to initialize the SWBus and register with **control**, just like the server process did. Our client registers as process "*simpleClient*".

SIMPLECLIENT

```
#include <swbus.h>

int main()
{
    char* clientName = "simpleClient";

    if ( SbInit(clientName,NULL,NULL) != SbCOK )
    {
        printf( "SbInit failed for process %s\n",clientName );
        exit( 1 );
    }
```

4.19 Establishing a connection to a remote process

The client can now connect to the server process. Connections to other processes are established by creating an instance of the class **SbCCRemote** or a subclass of it. An instance of **SbCCRemote** represents the remote process in the local process. Creating such an instance can be accomplished using ***SbCreate***, as for any other object, but a convenient function called ***SbOpen*** is simpler to use and is suitable in most situations.

SIMPLECLIENT (continued)

```
char* serverName = "simpleServer";
SbTSTI theServer;
printf( "Connecting to remote process %s ...\n", serverName );
theServer = SbOpen( SbCCRemote, serverName, "~remote", SbCWait );
printf( "Connection to process %s established\n",
        SbName(theServer) );
```

SbOpen takes four arguments. The first is the STI of the class to use locally to represent the remote process, and it must be **SbCCRemote** or a subclass of **SbCCRemote**. Subclasses of **SbCCRemote** can be created by the programmer using ***SbSub*** as usual (see Chapter 5 for an example). The second argument is the name of the process to connect to. This name must be the same as the name used in the call to ***SbInit*** by the remote process; we have used the name "*simpleServer*" for both. The third argument is the name of the class to be used by the remote process to represent our process. We use the name "*~remote*" which is the name of the class with the STI **SbCCRemote**. The last argument is the mode of the connection operation. **SbCWait** means that ***SbOpen*** should wait and not return until the connection is established. **SbCContinue** is the other option, meaning that ***SbOpen*** will just initiate a connection and return immediately. In such a case, the connection will be established at a later time, and the user can be informed about that via a special method called "*~notify*". Details about this can be found in Chapter 5 of this document and in the SWBus Reference Manual.

SbOpen returns a reference to the remote process. This reference is an STI, just like any other object.

We used **SbCWait** as our connection mode, so we know that the connection is established when ***SbOpen*** returns.

4.20 Accessing a variable in a remote process

To access a variable in a remote process we need to retrieve a reference to it. A reference to an object in a remote process is called a proxy.

SIMPLECLIENT (continued)

```
SbTSTI serverVar;
serverVar = SbId( theServer, "myInt" );
```

A proxy is created using ***SbId***. ***SbId*** always returns an attribute of an object, so calling ***SbId*** with a remote process object as first argument will make it return an object from the remote process. As an object always resides in a specific process, ***SbId*** returns a proxy to the remote object. If the object dos not exist in the remote process, ***SbId*** will return **SbCError**.

Our example makes ***SbId*** return a proxy to the integer object called *myInt*, which we created in the server process in Section 4.5.

Updating the value of a remote variable

Updating a variable value in a remote process is a frequently used operation. Using the SWBus, it is almost as simple as updating the value of a local variable.

SIMPLECLIENT (continued)

```
i = 49;
SbPut( serverVar, &i );
SbFlush( SbCPLocal, theServer );
printf( "Value %d assigned to variable %s in process %s\n",
        i, SbName(serverVar), SbName(theServer) );
```

ServerVar represents the integer object *myInt* in process *simpleServer*, and we intend to assign the value 49 to it. The function to be used is again ***SbPut***, just like for local objects. However, sending single variable values through the network would be unacceptably slow in applications that update several thousand variables, so ***SbPut*** just puts the value into a local buffer for later transfer. If this buffer gets full, ***SbPut*** will flush it. ***SbFlush*** is used to indicate that it is time to transfer the values put into the buffer.

SbFlush takes two arguments, the first is the process from which values should be flushed, and the second is the process to accept them. **SbCNull** can be used for the last argument to indicate that values should be flushed to all processes.

Retrieving the value of a remote variable

Retrieving the value of a remote variable is accomplished using ***SbGet***, just like for local variables.

SIMPLECLIENT (continued)

```
SbGet( serverVar, &j );
printf( "Value %d retrieved from variable %s in process %s\n",
        j, SbName(serverVar), SbName(theServer) );
```

SbGet is rarely used for remote variables, so the values are fetched one at a time.

4.21 Calling a function in a remote process

The principle for calling functions is very much the same as for accessing objects. First a proxy for the function must be created, then the operation should be as similar to its local equivalent as possible.

SIMPLECLIENT (continued)

```
SbTSTI serverFunc, localInt;
serverFunc = SbId( theServer, "myFunc" );
localInt = SbCreate( SbCPLocal, SbCCInteger, "localInt",
                     SbCNull, NULL );
i = 56;
SbPut( localInt, &i );
SbCall( serverFunc, localInt, localInt );
SbGet( localInt, &j );
printf( "%s(%d) = %d\n", SbName(serverFunc), i, j );
```

A proxy to the function *myFunc* created in Section 4.8 is extracted using ***SbId***. Note that the proxy represents the instance of the function, not the function class. A value is put into the local object *localInt* and the remote function is called using the local object as the actual parameter. The output value is then retrieved from the local object and the result is written to the screen.

The complete example also includes another remote function call. The function named *terminate* in the server process is called to make the server stop, so that both processes are terminated normally by the example code. The function *terminate* was declared in the server process without input or output arguments. Here is how it is called from the client process.

SIMPLECLIENT (continued)

```
SbCall( SbId( theServer, "terminate" ), SbCNull, SbCNull );
```

4.22 Closing the connection to a remote process

We have previously stated that the connection to the remote process was established by creating an instance of the class **SbCCRemote**, and demonstrated the convenience function ***SbOpen***. The connection is closed by calling ***SbDelete***, like you delete any other object.

SIMPLECLIENT (continued)

```
    SbDelete( theServer, SbCNull );
```

Finally, ***SbExit*** is called before exiting our client.

SIMPLECLIENT (continued)

```
    SbExit();
    return 0;
} /* End of main */
```

4.23 How to compile and run the example programs

Instructions for how to compile and run the example can be found in the *README* file of the *examples* subdirectory of the SWBus installation directory.

Note that you probably don't have write permission to the *examples* directory, so you should copy the source code and makefile to your own directory before compiling.

Good luck !

5 Advanced Programming

This chapter introduces a number of advanced features of the Software Bus by way of a detailed example application.

5.1 Introduction

This chapter begins by looking at the following topics, related to working with SWBus objects in general:

- checking object status
- direct manipulation of object data areas
- anonymous objects

The following advanced features of the SWBus are then introduced by studying an example application:

- the class **SbCCRemote**
- the class **SbCCPeriodic**
- the class **SbCCIo**
- the class **SbCCGroup**
- linking objects with ***SbLink***
- customising ***SbLoop*** with ***SbSetLoop***

The example application simulates a relationship between a chocolate factory and one or more customers (shops). A detailed description of it can be found in Appendix C, along with the source code and instructions on how to compile and run the processes.

5.2 Checking object status

It is often important to check that a SWBus object is safe to use, in particular if an object is known to be used by several processes.

Examples of situations where it would be wise to check an object's status include:

- A process stores the STI of an object in a variable after using it the first time, but the object may no longer exist the next time it is needed.
- A call to a SWBus API function fails.

Most SWBus functions return values that can be checked to verify that they performed their tasks correctly. If the value returned is **SbCError** then the operation was unsuccessful.

When working with SWBus objects, the macros ***SbDGood*** or ***SbDBad*** should be used. They both take the STI of a SWBus object as input and return a Boolean value that indicates whether or not the assertion that the STI was 'good' or 'bad' is true.

A typical example, taken from the ***f_connect*** function in the factory process, is shown below:

FACTORY

```
bool f_connect( SbTSTI remoteProcessId )
{
   SbTSTI remoteLockObject;

   remoteLockObject = SbId( remoteProcessId, "Lock" );

   if ( SbDGood( remoteLockObject ) )
   {
      ...
   }
   ...
}
```

In this example, the *remoteLockObject* returned by ***SbId*** is tested before it is used. The ***SbDGood*** macro tests that the value is neither **SbCError** nor **SbCNull**. This is more convenient, and easier to read, than specifically testing that the value is nethier **SbCError** nor **SbCNull** directly.

To test if an object, that is known to have existed, has been deleted, ***SbDDead*** can be used. This is particularly useful when working with stored STIs for remote object proxies, because ***SbDGood*** and ***SbDBad*** can only indicate that the STI is valid (is or is not **SbCNull** or **SbCError**) and not whether or not the proxy to the remote object still exists.

In most of the example code in this document, tests such as these have been kept to a minimum, to keep the code concise. In practice, you should always test an object before using it if there is

any possibility that it might be unsafe to use. Testing a SWBus object to see whether it is 'good' or 'bad' can be considered to be the SWBus equivalent of testing for **NULL** pointers in C or C++.

5.3 Direct manipulation of object data areas

It is more efficient, and often more convenient, to manipulate an object's data area directly rather than by using ***SbGet*** and ***SbPut*** to modify the values of it's members.

The SWBus function ***SbData*** takes the STI of a local SWBus object as input and returns a pointer (**void***) to the object's data area. ***SbData*** cannot be used with a proxy that represent an object in a remote process, as the pointer returned would have no meaning in the context of the local process.

Since the data area is in the same format used by C for data structures, it is often convenient to define a C **struct** that has the same data layout as a SWBus object and to cast the **void*** returned by ***SbData*** to the C structure. The structure can then be used as a template to modify the data directly.

Examples of the use of ***SbData*** can be found in Chapter 4 (in Section 4.10 on adding constructors and destructors to SWBus classes) and in the Sections 5.6 and 5.7 on creating instances of the classes **SbCCPeriodic** and **SbCCIo**, later in this chapter.

The SWBus function ***SbOffset*** is typically used in conjunction with ***SbData***. ***SbOffset*** can be used to retrieve the offset of a member of a class. Using pointer arithmetic it is possible to access the data of an object member directly without first casting the data area of the entire object to a C structure. This is particularly useful if the instances of a particular class always have a minimum set of members that need to be accessed, but where the definition of the class might change, so that the exact layout of the objects' data areas is not constant.

Note that when data structures are manipulated directly, the SWBus has no way of verifying that what is done is legitimate. Extreme care must therefore be taken so that the memory structures are not corrupted by programming errors!

The SWBus does not update objects that are linked to an object that is modified in this manner, as it is not aware that any modifications have been made. ***SbSend*** (followed by ***SbFlush***) must be used to propagate the value of the object to all objects that subscribe to it. This is different to the behaviour of ***SbPut***, which can be viewed as a combined update and send operation.

5.4 Using anonymous objects

An anonymous object can be created by supplying an empty string (or **NULL**) as the object's name to ***SbCreate*** when it is instantiated.

In the example below, an anonymous instance of the integer class is created and deleted:

ANONYMOUS OBJECT EXAMPLE

```
SbTSTI anonymousInteger;
char*  objectName = NULL;

anonymousInteger = SbCreate( SbCPLocal, SbCCInteger,
                             objectName, SbCNull, NULL );

SbDelete( anonymousInteger, SbCNull );
```

Although it is good practice to name objects, there is little point in naming an object whose name will never be used. It is often sufficient to simply store an object's STI when it is created and use it to work with the object later.

Using an object's name to look up an STI is inefficient compared with storing and reusing an STI directly. Calling ***SbId*** more often than necessary should be avoided if maximum performance is an important issue.

An example of where it is convenient to use anonymous objects is when creating objects to be used as parameters. Normally, a parameter object has a very short life; it is created, used once, and is then deleted. It is not necessary to look up it's name, so it does not need one.

In general, an object can be anonymous if it's STI is stored when it is created and if there is no need to retrieve it's STI using ***SbId*** later. Therefore, any object that is to be accessible to a process other than the one that created it should not be anonymous (because its name will be needed to find it).

5.5 The class SbCCRemote

Each time a process opens a connection to a remote process and each time a remote process connects, the SWBus creates an instance of the class **SbCCRemote** to represent the remote process.

Like all other SWBus classes, it is possible to create subclasses of **SbCCRemote** to customise the handling of different kinds of processes.

When a process calls ***SbOpen*** to connect to a remote process, it specifies what kind of process it is. The remote process uses this information when it creates an instance of **SbCCRemote** (or a subclass of it) to represent the process.

The simplest possible code for this is shown below. It is a cut-down version of the code that creates the subclasses in ***f_initialise*** and ***c_initialise***, in the customer and factory example processes.

FACTORY

```
SbTSTI customerClass;
...

customerClass = SbSub( SbCCRemote, "CustomerClass", SbCNull );

...
```

CUSTOMER

```
SbTSTI factoryClass;
char* factoryName = ...
...

factoryClass = SbSub( SbCCRemote, "FactoryClass", SbCNull );

SbOpen( factoryClass, factoryName, "CustomerClass", SbCContinue );

...
```

The customer process takes the initiative to connect to a factory by calling ***SbOpen.***

SbOpen creates a local object called **factoryName**, which is an instance of the class **factoryClass** (which in turn is a subclass of **SbCCRemote**). It then attempts to connect asynchronously (**SbCContinue**) to a remote process called **factoryName**, declaring itself as a process of the named type "**CustomerClass**".

When the factory receives the incoming connection, it creates an instance of it's local representation of the class called "**CustomerClass**" to represent the remote customer process. The instance is given the same name as the customer process that it represents (as set when the customer process called **SbInit**)

Note that if we had passed "**~remote**" (the name of **SbCCRemote**) instead of "**CustomerClass**" to ***SbOpen***, then the factory would have created an instance of **SbCCRemote**.

To establish a connection synchronously, **SbCContinue** can be replaced with **SbCWait**.

A subclass of **SbCCRemote** can be provided with members of its own in the same way that members are added to other classes. This is particularly useful for storing information about a process in the process object itself, making the data easy to locate and manage. For example, if it is desirable to maintain a table of values of some sort for each process, then a table can be created in the subclass' constructor function, a pointer to the table can be put into an attribute in the object for easy retrieval, and the table can be deleted when the subclass' destructor is called.

SbCCRemote has two special methods that can be overridden using ***SbAdd***: ***~update*** and ***~notify***.

The ~update method

The ***~update*** method is called whenever objects are updated by the remote process.

This is useful if it is necessary to perform some action whenever an update occurs, such as updating a display to show new values.

The ~notify method

Whenever the state of the connection to the remote process changes, the ***~notify*** method is called.

The input parameter of ***~notify*** is a SWBus object containing a bit mask. Bitwise operations are used to test the value , since notification of the occurrence of more than one event can be contained in the parameter.

The ***~notify*** method enables a process to be designed to cope with the following events:

- **SbCBPSConnecting** - Trying to connect to the remote process.
- **SbCBPSConnected** - Connection with the remote process has been established.
- **SbCBPSDisconnect** - The remote process disconnected.
- **SbCBPSRequest** - The remote process requests a connection. (received with **SbCBPSConnected** or **SbCBPSDisconnect**).
- **SbCBPSWaiting** - Waiting in internal loop for a connection to the remote process to be established. (received with **SbCBPSConnecting** for synchronous connections).
- **SbCBPSBroken** - Connection to the remote process is broken.
- **SbCBPSTimeout** - Given up trying to connect to a remote process.
- **SbCBPSIncompVersion** - Trying to connect to a SWBus process with an incompatible version of the SWBus library.

The file connect.c in the examples directory of the SWBus installation directory serves as a complete referance on how to set up a ***~notify*** method.

The complete function (***f_notify***) and subclass declaration for the factory process (in ***f_initialise***) is shown below:

FACTORY

```
bool f_connect( SbTSTI remoteProcessId );
void f_disconnect( SbTSTI remoteProcessId );

...


SbTSTI f_notify( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
  SbTType notification;

  SbGet( in, &notification );

  if ( notification & SbCBPSConnected )
      f_connect( o );

  if ( ( notification & SbCBPSDisconnect )
      || ( notification & SbCBPSBroken ) )
      f_disconnect( o );

  return out;
}

...


bool f_initialise( ... )
{
  SbTSTI customerClass;
  SbTSTI notifyFunction;

  ...

  notifyFunction = SbDeclare( "FactoryNotify", SbCCType,
                              SbCNull, &f_notify, SbCLC );

  customerClass  = SbSub( SbCCRemote, "CustomerClass", SbCNull );
  SbAdd( customerClass, notifyFunction, "~notify", NULL );

 ...

}
```

The *in* argument, which is passed to the ***~notify*** function, contains a variable of type **SbTType**. It holds information about the change of the state of the connection that triggered the invocation of the ***~notify*** method.

In ***f_notify***, ***SbGet*** is used to extract the value of the *in* argument. The value (**notification**) is then tested to see if the remote process has connected and then to see if the remote process has discon-

nected (gracefully or otherwise). Anything else is ignored. This information is used to call either ***f_connect*** and/or ***f_disconnect*** (see the Section 5.8 on the 'group' class, later in this chapter, to see what ***f_connect*** and ***f_disconnect*** do).

The ~state attribute

SbCCRemote has an attribute called **~state** that is useful if it is necessary to check the state of a remote object. The value of the **~state** attribute must be extracted using ***SbData***.

In a client-server situation, such as that in the example application, when the server dies, the client attempts to re-establish the connection. The state of the server's remote object in the client will first be **SbCBPSBroken**, then **SbCBPSConnecting**, and finally **SbCBPSConnected** (if it succeeds in connecting).

When a client dies, the server deletes the remote object that represents the client. A server process can test the STI of a client process object to see if has been deleted using ***SbDDead***.

5.6 The class SbCCPeriodic

The class **SbCCPeriodic** is used to create function handlers that are called on a regular basis from the SWBus' main loop.

In the example application, instances of **SbCCPeriodic** are used in each process to initiate new time periods at predefined intervals, thus regularly regaining control from the main loop to perform a number of tasks before returning to it.

An instance of **SbCCPeriodic** is created using ***SbCreate*** with an instance of the class **SbCCPeriodicParameter** as input. The periodic parameter object contains the initial values that are passed to the constructor of the periodic object.

The function ***SbData*** can be used to retrieve a pointer to the parameter object's data area. The pointer can be cast to the C structure **SbTPeriodicParameter**. The the values for the parameter object's attributes can then be set by modifying a C structure rather than by using ***SbPut*** calls.

The steps involved to create a periodic object are:

1) Create an instance of **SbCCPeriodicParameter** and assign values to its attributes, one of which is a pointer to a SWBus-callable handler that will be called at regular intervals.
2) Create an instance of **SbCCPeriodic** with the parameter object as the input parameter to its constructor.
3) Delete the parameter object (optional).

Code taken from the factory process will now be examined to see how a periodic object is created.

The code below is taken from the function ***f_initialise***:

FACTORY

```
/* Globally defined: */
/* SbTSTI f_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out); */
/* int32 timeInterval = 2000; */

    ...

    SbTSTI periodicIn  = SbCNull;
    SbTSTI periodicOut = SbCNull;
    int32  periodicCount  = -1;
    int32  periodicPeriod = -1;
    SbTSTI periodicParameter;
    SbTPeriodicParameter* periodicParameterData;

    ...

1.  periodicParameter = SbCreate( SbCPLocal, SbCCPeriodicParameter,
                                  NULL, SbCNull, NULL );

2.  periodicParameterData
       = (SbTPeriodicParameter*)SbData( periodicParameter );

3.  periodicParameterData->interval = timeInterval;
4.  periodicParameterData->count    = periodicCount;
5.  periodicParameterData->period   = periodicPeriod;
6.  periodicParameterData->term     = NULL;
7.  periodicParameterData->func     = f_periodic;
8.  periodicParameterData->inParam  = periodicIn;
9.  periodicParameterData->outParam = periodicOut;
    ...
```

After the periodic parameter object has been created using ***SbCreate*** (1.), a pointer to the **SbTPeriodicParameter** data area is retrieved (2.), which is then used to assign values to its attributes, which will be used to create a periodic object.

The members of the **SbTPeriodicParameter** structure are:

- **interval** (3.) - The time in milliseconds between each call to the handler.
- **count** (4.) - The number of times the handler should be called before the periodic object suspends automatically. If the value is `-1` then there is no limit to how many times the handler can be called by the periodic object.
- **period** (5.) - The time in milliseconds after which the periodic object should automatically stop calling the handler. If it is `-1` then the periodic object will have no time limit.
- **term** (6.) - The termination function. When the limit specified in either the **count** member or the **period** member is reached, this termination function is called. If it is set to **NULL** then no function will be called when this occurs. This function is not called when a periodic object is otherwise suspended or deleted.

- **func** (7.) - The periodic handler function. This is a SWBus-callable function that will be called regularly while the periodic object is active.

inParam (8.) - The STI of a SWBus object that is passed to the handler each time it is called by the periodic object. In the example it is **SbCNull**.

outParam (9.) - The STI of a SWBus object that is returned from the handler each time it is called by the periodic object. In the example it is **SbCNull**.

The periodic object is then created using the STI of the periodic parameter object as input. The parameter object is subsequently deleted. The creation of a periodic object is complete:

FACTORY

```
SbTSTI periodic;
SbTSTI periodicParameter;

...

periodic = SbCreate( SbCPLocal, SbCCPeriodic,
                     NULL, periodicParameter, NULL );

SbDelete( periodicParameter, SbCNull );
...
```

Note that both the periodic parameter and the periodic objects are anonymous in this example. There is normally no need to name an instance of the **~periodicParameter** class because it will usually be deleted after it has been used to initialise an instance of the **~periodic** class. However, it is not strictly speaking necessary to delete a parameter object after it has been used if it is going to be reused to initialise another periodic object.

The periodic object created in the example above will call the function ***f_periodic*** every **timeInterval** milliseconds.

The ***f_periodic*** function in the factory process is shown below as an example of a periodic handler function:

FACTORY

```
SbTSTI f_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   f_produceChocolates();
   f_distributeChocolates();

   return out;
}
```

Methods in **SbCCPeriodic** can be invoked in a periodic object to suspend calls to the handler, to reactivate calls to the handler, and to modify the time interval between calls.

Examples of invocations of methods provided by **SbCCPeriodic** are shown below:

PERIODIC EXAMPLE

```
SbTSTI periodic;
SbTSTI newInterval;
int ms = 1000; /* 1000 milliseconds */

...

/* suspend the periodic object */

SbCall( SbId( periodic, "~suspend" ), SbCNull, SbCNull );


/* modify the time interval */

newInterval = SbCreate( SbCPLocal, SbCCInteger, NULL, SbCNull, &ms);
SbCall( SbId( periodic, "~setInterval" ), newInterval, SbCNull );
SbDelete( newInterval, SbCNull );


/* restart the periodic object */

SbCall( SbId( periodic, "~restart" ), SbCNull, SbCNull );
```

Note that the code above is not taken from the example application but is provided to illustrate how the periodic object's methods should be invoked. The Reference Manual contains detailed information about the use of these methods.

When a periodic object is no longer required, it can be deleted using ***SbDelete***. If the objects used as input and output parameters to the handler are no longer needed then they should also be deleted using ***SbDelete***.

Any number of periodic objects can be created and used to perform different tasks.

5.7 The class SbCCIo

The class **SbCCIo** is used to create function handlers that are called as a result of activity associated with one or more file descriptors. This is useful when combining SWBus with other socket based communication systems or when creating keyboard handlers. The SWBus uses io objects internally to handle, amongst other things, communication between processes.

In the example that will be examined in this section, an instance of **SbCCIo** is responsible for calling a function when keyboard activity takes place. Note that the example will not work on Windows platforms, as no file descriptor can be obtained for stdin.

An instance of **SbCCIo** is created using ***SbCreate*** with an instance of **SbCCIoParameter** as input. The parameter object contains the initial values that are passed to the constructor of the io object.

The function ***SbData*** can be used to retrieve a pointer to the parameter object's data. The pointer can be cast to the C structure **SbTIoParameter**. The the values for the parameter object's attributes can then be set by modifying a C structure rather than by using ***SbPut*** calls.

The steps involved to create an io object are:

1) Specify the condition for the activity on the file descriptor that will trigger the calling of a function.
2) Create a file descriptor set that can be used to specify which file descriptors the io object should monitor.
3) Create an instance of **SbCCIoParameter** and assign values to it's attributes, one of which is a pointer to a SWBus-callable handler that will be called when the trigger conditions of the io object are met.
4) Create an instance of **SbCCIo** with the parameter object as the input parameter to it's constructor.
5) Delete the parameter object (optional).

Code taken from the factory process will now be examined to see how an io object is created.

This code below is taken from the function ***f_initialise***:

FACTORY

```
/* Globally defined: */
/* SbTSTI f_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out); */

    ...

    int32  ioCondition;
    fd_set *ioFdSet;
    SbTSTI ioIn  = SbCNull;
    SbTSTI ioOut = SbCNull;
    SbTSTI io;
    SbTSTI ioParameter;
    SbTIo* ioParameterData;
    ...

    ioCondition = SbCFDREAD;
    ioFdSet     = (fd_set*)malloc( sizeof(fd_set) );
    FD_ZERO( ioFdSet );
    FD_SET( STDIN_FILENO, ioFdSet );

1.  ioParameter = SbCreate( SbCPLocal, SbCCIoParameter,
                            NULL, SbCNull, NULL );
2.  ioParameterData = (SbTIoParameter*)SbData( ioParameter );
3.  ioParameterData->fdc      = ioCondition;
4.  ioParameterData->fds      = ioFdSet;
5.  ioParameterData->func     = f_io;
6.  ioParameterData->inParam  = ioIn;
7.  ioParameterData->outParam = ioOut;

    io = SbCreate( SbCPLocal, SbCCIo, NULL, ioParameter, NULL );
```

After the parameter object has been created using ***SbCreate*** (1.), a pointer to the **SbTIoParameter** data area is retrieved (2.), which is then used to assign values to its attributes, which will be used to create an io object.

The members of the **SbTIoParameter** structure are:

- **fdc** (3.) - The condition for the activity on the associated file descriptor set that will trigger the io object. Legal values are **SbCFDREAD**.
- **fds** (4.) - The file descriptor set.
- **func** (5.) - The io handler function. This is a SWBus-callable function that will be called when the io object's trigger condition is met.

inParam (6.) - The STI of a SWBus object that is passed to the handler each time it is called by the io object. In the example it is **SbCNull**.

outParam (7.) - The STI of a SWBus object that is returned from the handler each time it is called by the io object. In the example it is **SbCNull**.

Note that both the io parameter and the io objects in the example are anonymous. There is normally no need to name an instance of **SbCCIoParameter** because it will usually be deleted after it has been used to initialise an instance of **SbCCIo**. However, it is not strictly speaking necessary to delete a parameter object after it has been used if it is going to be reused to initialise another io object.

The ***f_io*** function that is called when keyboard activity is detected by the factory process is shown below as an example of an io handler function:

FACTORY

```
SbTSTI f_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   char buffer[64];

   scanf( "%s", &buffer );

   if ( strcmp( buffer, "quit" ) == 0 )
      f_quit();
   else
      if ( strcmp( buffer, "query" ) == 0 )
         f_queryWarehouse();

   return out;
}
```

When an io object is no longer required, it can be deleted using ***SbDelete***. If the objects used as input and output parameters to the handler are no longer required then they should also be deleted using ***SbDelete***.

Any number of io objects can be created and used to perform different tasks.

5.8 The class SbCCGroup

The class **SbCCGroup** is a generic class that can be used to store collections of SWBus objects that can be accessed collectively using single common operations.

The following SWBus operations can be performed on groups of objects:

- ***SbCall*** - calling a group of functions results in each function in the group being executed and a group of return values being returned.
- ***SbId*** - returns a group of attributes if the parent argument in the call is a group.
- ***SbPut*** - puts values into each object in a group when called for a group.
- ***SbFlush*** - flushes values from all processes in the group.

SbCCGroup is a sub-class of **SbCCList**. This means that list class methods can also be used on groups of objects.

In the example application, the factory process maintains a group of customer objects. The STI for the group is stored in a global variable when it is created in the function ***f_initialise***. When a customer process connects or disconnects, a customer object is included in or withdrawn from the group as appropriate.

Example code from the factory process is shown below:

FACTORY

```
SbTSTI customerGroup = SbCNull; /* Global Variable */

...

f_initialise( ... )
{
  ...

  customerGroup = SbCreate( SbCPLocal, SbCCGroup, NULL, SbCNull, NULL );

  ...
}


bool f_connect( SbTSTI remoteProcessId )
{
  ...

  SbInclude( customerGroup,
             SbList( SbCCGroup, remoteProcessId, SbCNull ) );
  ...
}


void f_disconnect( SbTSTI remoteProcessId )
{
   SbWithdraw( customerGroup,
               SbList( SbCCGroup, remoteProcessId, SbCNull ) );
   ...
}
```

The functions ***f_connect*** and ***f_disconnect*** are called when a remote process connects and disconnects respectively, as a result of a test in the function ***f_notify***.

Note that when calling ***SbInclude*** and ***SbWithdraw,*** it is necessary to use ***SbList*** to create a temporary list object to include/withdraw the object in/from the group.

An example of the traversing of the contents of a group can be found in the function ***f_distributeChocolates*** in the factory process. The function ***SbFirst*** followed by calls to ***SbNext*** is used to iterate over the group. The last object in a list is always an **SbCNull** object so a test that an object is not `SbCNull`, either directly or by using either ***SbDGood*** or ***SbDBad,*** can be used to specify the stopping condition for the iteration.

The function ***f_distributeChocolates*** is shown below:

FACTORY

```
void f_distributeChocolates()
{
   SbTSTI theCustomer;

...

   theCustomer = SbFirst( customerGroup, SbCNull );

   while ( SbDGood( theCustomer ) && !SbDDead( theCustomer ) )
   {
      ...

      theCustomer = SbNext( customerGroup );
   }

   ...
}
```

If the factory's stock of chocolates is too large, it can tell it's customers that chocolates should be sold at a discount price, in order to sell more per time period and reduce the amount of chocolates in stock.

When it initialises, each customer process declares a function (***c_discount***) to the SWBus with the name "**discount**". This function needs to be called for each customer when the factory initiates a discount period. To do this, the factory process' ***f_discount*** function calls ***SbId***, with the customer group as parent attribute, for the (function) object called "**discount**". This results in a group of STI values. ***SbCall*** is then performed on this group to call each function in the group. The input parameter will be passed to each function that is called.

The function ***f_discount*** is shown below:

FACTORY

```
bool discountPeriod; /* global variable */

...

void f_discount( bool startDiscountPeriod )
{
   SbTSTI discountFunctions;
   SbTSTI inputParameter;

   if ( SbHead( customerGroup ) == SbCNull ) /* empty group? */
      return;

   discountFunctions = SbId( customerGroup, "discount" );

   if ( SbDGood ( discountFunctions ) )
   {
      inputParameter = SbCreate( SbCPLocal, SbCCInteger,
                                 NULL, SbCNull, NULL );
      SbPut( inputParameter, &startDiscountPeriod );
      SbCall( discountFunctions, inputParameter, SbCNull );
      SbDelete( inputParameter, SbCNull );
      discountPeriod = startDiscountPeriod;
   }
}
```

Note that the head of the customer group is checked before proceeding to verify that the group is not empty. There is little point in attempting to retrieve a group of function object STI values from an empty group.

5.9 Object linking

In some situations it is useful to have a local copy of a remote object in order to reduce the amount of communication between processes on different machines and to enable a distributed system to be able to cope with a system failure on a remote machine. However, we also want the values of the objects to be the same without us having to call ***SbPut*** to copy the value of the source object to each target object. To do this, the SWBus function ***SbLink*** can be used. Any objects, either local or remote, can be linked together as long as they have the same data format.

SbLink has three parameters. The first parameter specifies the *source* object and the second parameter specifies the *target* object. The objects can be in the same process or in different processes, which may also be on different machines. The third parameter is the link *mode*, which is used to specify whether or not the value of the *source* object should be sent to subscribers of the *target* object. If the values are to be redistributed then the *mode* should be **SbCBPPutRedistribute**, otherwise it should be **SbCBPNull**.

When ***SbPut*** or ***SbSend*** are called for the *source* object, the value will be transferred to the *target* object. If the *target* object is in a remote process then the actual data transfer will not take place until ***SbFlush*** is called.

In both the customer and the factory process there is an object called "**Lock**" of type **SbCCInteger**. This object is created in the ***f_initialise*** and ***c_initialise*** functions and the STI of the object is stored globally as **lockObject** in each process. However, the 'lock' object in the factory process is provided with a value, a Boolean **lock**. When a customer process connects to the factory, a link is made from the 'lock' object in the factory to the 'lock' object in the remote customer process. The code in the functions ***f_initialise***, ***f_connect***, and ***c_initialise*** is shown below to show how two objects are linked together:

FACTORY

```
SbTSTI lockObject = SbCNull; /* Global Variable */
bool lock = false; /* Global Variable */

...


bool f_initialise( ... )
{

   ...

   lockObject = SbCreate( SbCPLocal, SbCCInteger, "Lock",
                          SbCNull, &lock );
   ...

}


bool f_connect( SbTSTI remoteProcessId )
{
   SbTSTI remoteLockObject;

   ...

   remoteLockObject = SbId( remoteProcessId, "Lock" );

   if ( SbDGood( remoteLockObject ) )
      SbLink(lockObject, remoreLockObject, SbCBPNull );

   ...

}
```

CUSTOMER

```
SbTSTI lockObject = SbCNull; /* Global Variable */

...


bool c_initialise( ... )
{

   ...

   lockObject = SbCreate( SbCPLocal, SbCCInteger, "Lock",
                          SbCNull, SbCNull );
   ...

}
```

The link from the 'lock' object in the factory process will automatically be removed when the customer process disconnects, so there is no need to call ***SbUnlink*** in ***f_disconnect***.

The 'lock' object is used to prevent customer processes from updating their inventories (by selling chocolates) while the factory process is also updating their inventories (when distributing chocolates).

If the factory has chocolates in storage that can be distributed, the function ***f_distributeChocolates*** sets the 'lock' to **true** before distribution begins, and to **false** after distribution has been completed. In both cases the change is sent to the customer processes immediately. Remember that the **lockObject**'s value is the value of the variable **lock**, as specified when the **lockObject** was created in ***f_initialise***.

The relevant code for this is shown below:

FACTORY

```
...

lock = true;
SbSend( lockObject );
SbFlush( SbCPLocal, customerGroup );

...

/* Distribute chocolates to customer processes */

...

lock = false;
SbSend( lockObject );
SbFlush( SbCPLocal, customerGroup );

...
```

The function ***c_sellChocolates***, in the customer process, checks whether it has a valid local 'lock' object STI and, if the STI is valid, gets the object's value and checks it in order to decide how to proceed. The relevant code is show below:

CUSTOMER

```
...

if ( SbDGood( SbGet( lockObject, &lock ) ) )
  if ( lock == true )
  {
     printf( "Shop is closed while restocking...\n" );
     return;
  }

...
```

The function ***SbSubscriptions*** can be used to retrieve the STIs of the objects that are linked ("subscribed") to an object. ***SbSubscriptions*** returns a group that can be traversed as described in the Section 5.8 on the class **SbCCGroup** earlier in this chapter.

If instances of two identical classes in different processes are to be linked together and it is possible that the definition of the classes might change while the process is running, then it is also necessary to link the class objects.

If two classes, in separate processes, are linked together then, when the *source* class object is modified, ***SbSend*** must be called to buffer the changes, followed by ***SbFlush*** to send them over to the *target* class, otherwise the modifications will not be distributed. This is because, unlike ***SbPut***, ***SbAdd*** does not explicitly distribute modifications made to classes.

To remove a link between a *source* and a *target* object, ***SbUnlink*** must be called. If, however, a link exists between two objects in different processes, and one of the process dies, then any links to objects in the dead process will no longer be valid, and will, therefore, cease to exist. Should the dead process be restarted and a new connection be established, then the objects must be relinked using ***SbLink***.

Links between identical objects can considerably reduce the amount of communication between processes, because objects can be accessed locally and only modifications are transfered between processes. Any situation where a process uses ***SbGet*** to frequently access the value of a remote object, whose value changes less frequently, can usually be handled more efficiently by introducing a local copy of the object and linking it to the remote object. ***SbGet*** can then be used to retrieve the value of the local object, avoiding unnecessary network activity.

5.10 Customising SbLoop with SbSetLoop

Although io and periodic handlers can be used to regain control from the main loop, it is sometimes necessary to completely override the default loop, in order to exercise even greater control. This is often the appropriate when existing applications with complex main loops are modified to use the SWBus.

An alternative main loop must provide the same basic functionality as the default loop. In other words, it must execute all pending periodic functions and io handlers, and it must receive data from remote processes and execute local functions called by remote processes.

The function ***SbSetLoop*** is used to replace the default loop with a user-defined function. Calling ***SbLoop*** then passes control to the user-defined function.

A simple example is shown below:

SbSetLoop Example

```
void UserLoop( int32 semaphore ); /* Defined later */

...

SbSetLoop ( UserLoop );

...

SbLoop();
```

SbSetLoop takes the user-defined function as input. The function should have just one input parameter, a semaphore, that is used by the SWBus to specify the loop's stopping condition.

The loop function needs to perform at least the following tasks in the order in which they are listed below:

1) Enter a **while** loop that uses ***SbIsActive*** to check if the stopping condition has been met. ***SbIsActive*** takes the semaphore passed to the loop function as input.

2) Call ***SbPeriodic***. ***SbPeriodic*** executes all pending periodic functions and returns the time to wait for the next periodic function to be called. It returns **NULL** if no periodic functions are pending. If ***SbPeriodic*** is unable to execute a periodic function in time then it shortens the interval to the next execution in order to get back into sync.

3) Use ***SbFDGet*** to get the SWBus file descriptor set currently in use by the local process. ***SbFDGet*** takes a file descriptor set as input (which it modified) and returns the size of the file descriptor set.

4) Call the ***select*** function with the file descriptor set from ***SbFDGet*** and a time-out no greater than the time to wait until the next periodic function is due to be called.

5) When the ***select*** function returns, call ***SbDispatch*** to check all active io objects, to see if any of them have file descriptor sets that match those passed to ***SbDispatch***. ***SbDispatch*** handles any pending io tasks before returning. Tasks include calling SWBus functions that are being executed by a remote process and receiving data from remote processes.

It is important to be aware that the SWBus can make recursive calls to ***SbLoop***, so care should be taken to avoid infinite recursion. Refer to the Reference Manual for further details about ***SbSetLoop***, ***SbIsActive***, ***SbPeriodic***, ***SbFDGet***, and ***SbDispatch***.

5.11 Summary

In this chapter we have looked at a number of advanced concepts that can be used to create powerful distributed software systems. We have looked at four of the SWBus' central classes in detail; **SbCCRemote**, **SbCCPeriodic**, **SbCCIo**, and **SbCCGroup**. We have discussed object linking, as well as various other aspects of working with objects. We have also looked at how to create a customised main loop.

Appendix A - API Quick Reference

This appendix contains a complete list of all SWBus API functions. Please refer to the Software Bus Reference Manual for detailed descriptions of each function.

A complete list of all predefined SWBus classes can be found in the Reference Manual, at the manual page for SbTSTI.

General

SbInit	Initialize the SWBus
SbExit	Exit the SWBus
SbSetMessageOutputCb	Register a callback function to set the output destination
SbSetMessageFilter	Set the wanted output
SbClearMessageFilter	Clear the wanted output

Objects

SbCreate	Create an object
SbDelete	Delete an object
SbPut	Put a value into an object
SbGet	Get a value from an object
SbSend	Distribute the value of an object
SbCopy	Copy the value of one object to another
SbName	Get the name of an object
SbClass	Get the class of an object
SbData	Get a pointer to the data area of an object
SbOffset	Get the value of the offset of an attribute
SbId	Get the STI of a named object

SbIds	Get the STI of several named objects
SbIndex	Get the STI of an element in an array
SbCast	Cast an object to another class
SbObject	Return a temporary object

Classes

SbSub	Create a new class
SbAdd	Add a member to a class
SbInstances	Return all of the instances of a class
SbAddr	Return a data area formatted according to a class description

Functions

SbDeclare	Create a new function class
SbCall	Call a function or method

Remote processes

SbOpen	Open a connection to a remote process
SbFlush	Transfer buffered data between processes
SbNonBlocking	Set the communication channel to a remote process nonblocking

SWBus loop

SbLoop	Enter the SWBus main loop
SbEndLoop	Terminate the SWBus main loop
SbSetLoop	Specify a user-defined SWBus main loop
SbIsActive	Test the stopping condition for the main loop
SbDataWaiting	Test for existence of unsent data
SbFDGet	Return the current SWBus file descriptor set of the local process
SbFDAnd	Perform a logical AND operation of two file descriptor sets
SbFDOr	Perform a logical OR operation of two file descriptor sets
SbFDSize	Calculate the size of a file descriptor set
SbSetFdChangeCb	Register a callback function for changes in the SWBus file descriptor set

SbPeriodic	Execute pending periodic functions
SbNextTimeout	gives the time to next periodic function
SbDispatch	Call SWBus functions if requested by remote processes

Object linking

SbLink	Create a one-way link between two objects
SbUnlink	Remove a link between two objects
SbSubscriptions	Return all links (subscriptions) to an object

Lists

SbList	Return a temporary list object
SbFirst	Return the first object in a list
SbNext	Iterate over a list of objects
SbHead	Get the first object in a list
SbTail	Return a list containing all elements after the first element
SbCrown	Add an object to the front of a list
SbBehead	Remove the first element of a list
SbInclude	Include members of a list in another list
SbWithdraw	Remove members of a list from another list

Appendix B - Basic Example

This is the complete source code used in Chapter 4.

simpleServer.c

```
/* SWBus application
 *
 * Application: simpleServer (BASIC EXAMPLE)
 *
 * File: simpleServer.c
 *
 * Author: Hakon Jokstad
 *
 * Date  : 10Mar96 HJo
 *
 */

#include <swbus.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


/* Number of elements in the SWBus array created later*/
#define NumArrayElements 5


/* A C struct matching the SWBus class "myClass" created later */
struct myStruct
{
   int   theInt;
   float theFloat;
   char* theString;
};


/* C prototypes for SWBus functions */
SbTSTI theFunction      ( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI printObjectValues( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI constructor      ( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI destructor       ( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI terminate        ( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );


int main()
{
   SbTSTI myInt;         /* A SWBus integer variable */
   SbTSTI funcClass;     /* A SWBus function class */
   SbTSTI myFunc;        /* A SWBus function instance */
   SbTSTI myClass;       /* A SWBus class */
   SbTSTI myObject;      /* A SWBus object of class myClass ... */
   SbTSTI printMet;      /* ... and its print method */
   SbTSTI param;         /* A parameter object used several times */
   SbTSTI arrayClass;    /* A SWBus array class */
   SbTSTI myArray;       /* A SWBus array */
   SbTSTI element;       /* An element of myArray */
   SbTSTI termFunc;      /* A function to terminate the server */
   SbTArrayParameter paramData;     /* C variable used by param */
   int i, j;             /* C variables to set and get SWBus object values */
   char*  serverName = "simpleServer"; /* The name of the server process */
```

```
/* Initialize SWBus. Register myself as process "simpleServer".
   Control Server host specified by environment variable CONTROLHOST,
   localhost used if CONTROLHOST is unset */

if ( SbInit( serverName, NULL, NULL ) == SbCError )
{
   printf( "SbInit failed for process %s\n", serverName );
   exit( 1 );
}


/* Create an integer object */
myInt = SbCreate( SbCPLocal, SbCCInteger, "myInt", SbCNull, NULL );

/* Put a value into the object and get it back */
i = 5;
SbPut( myInt, &i );
printf( "Value %d assigned to object %s\n", i, SbName(myInt) );
SbGet( myInt, &j );
printf( "Value %d retrieved from object %s\n", j, SbName(myInt) );


/* Create a function with integer input and output parameters */
funcClass = SbDeclare(NULL,SbCCInteger,SbCCInteger,theFunction,SbCLC);
myFunc    = SbCreate( SbCPLocal, funcClass, "myFunc", SbCNull, NULL );

/* Call the function */
i = 37;
SbPut( myInt, &i );
SbCall( myFunc, myInt, myInt );
SbGet( myInt, &j );
printf( "%s(%d) = %d\n", SbName(myFunc), i, j );


/* Create a class to match the C-struct myStruct */
myClass = SbSub( SbCCEmpty, "myClass", SbCNull );
SbAdd( myClass, SbCCInteger, "theInt", NULL );
SbAdd( myClass, SbCCFloat,   "theFloat", NULL );
SbAdd( myClass, SbCCString,  "theString", NULL );

/* Create a print method for the class */
funcClass = SbDeclare( NULL, SbCNull, SbCNull, printObjectValues, SbCLC );
SbAdd( myClass, funcClass, "print", NULL );

/* Create a constructor for the class */
funcClass = SbDeclare( NULL, SbCCInteger, SbCNull, constructor, SbCLC );
SbAdd( myClass, funcClass, "~construct", NULL );

/* Create destructor for the class */
funcClass = SbDeclare( NULL, SbCNull, SbCNull, destructor, SbCLC );
SbAdd( myClass, funcClass, "~destruct", NULL );

/* Create an instance of myClass, use value 2 for the constructor param */
i = 2;
param = SbCreate( SbCPLocal, SbCCInteger, NULL, SbCNull, &i );
myObject = SbCreate( SbCPLocal, myClass, "myObject", param, NULL );
SbDelete( param, SbCNull );

/* Call the print method of the object */
printMet = SbId(myObject,"print");
SbCall( printMet, SbCNull, SbCNull );

/* Delete the object */
SbDelete( myObject, SbCNull );
```

```
/* Create an array class */
   paramData.baseClass = myClass;
   paramData.count     = NumArrayElements;
   param = SbCreate(S bCPLocal, SbCCArrayParameter,NULL,SbCNull,&paramData );
   arrayClass = SbSub( SbCCArray, NULL, param );
   SbDelete( param, SbCNull );

   /* Create the array object */
   myArray = SbCreate( SbCPLocal, arrayClass, "myArray", SbCNull, NULL );

   /* Call the print method of each element in the array */
   for ( i = 0; i < NumArrayElements; i++ )
   {
      element = SbIndex( myArray, i );
      SbCall( SbId(element,"print"), SbCNull, SbCNull );
   }

   /* Delete the array */
   SbDelete( myArray, SbCNull );

   /* Create a function to terminate the server */
   funcClass = SbDeclare( NULL, SbCNull, SbCNull, terminate, SbCLC );
   termFunc  = SbCreate( SbCPLocal, funcClass, "terminate", SbCNull, NULL );

   /* Enter the main loop */
   printf( "%s entering main loop ...\n", SbName(SbCPLocal) );
   SbLoop();

   /* Leave the Software Bus */
   SbExit();

   return 0;
}


SbTSTI theFunction( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   int k;

   SbGet( in, &k );
   printf("Function %s.%s called with argument %d\n", SbName(o),SbName(f),k);
   k = 100 * k;
   SbPut( out, &k );
   return out;
}


SbTSTI printObjectValues( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   int   theInt;
   float theFloat;
   char* theString;

   SbGet( SbId(o,"theInt"),    &theInt );
   SbGet( SbId(o,"theFloat"),  &theFloat );
   SbGet( SbId(o,"theString"), &theString );
   printf( "Object %s: theInt = %d, theFloat = %f, theString = %s\n",
   SbName(o), theInt, theFloat, theString );
   return out;
}
```

```
SbTSTI constructor( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   struct myStruct* data = (struct myStruct*)SbData(o);

   if ( in == SbCNull )
   {  /* Default constructor */
      data->theInt = 1;
      data->theFloat = 1.1;
      data->theString = strdup( "default" );
     printf( "Default constructor of class %s called\n",SbName(SbClass(o)) );
   }
   else
   {  /* Constructor with integer input parameter */
      SbGet( in, &data->theInt );
      data->theFloat = 2.5 * data->theInt;
      data->theString = strdup( "some text" );
      printf( "Constructor of object %s called with value %d\n",
      SbName(o), data->theInt);
   }

   return out;
}


SbTSTI destructor( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   struct myStruct* data;

   printf( "Destructor of class %s called\n", SbName(SbClass(o)) );
   data = (struct myStruct*)SbData(o);
   free( data->theString );
   data->theString = NULL;
   return out;
}


SbTSTI terminate( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Terminate main loop */
   printf( "Server asked to stop\n" );
   SbEndLoop();
   return out;
}
```

simpleClient.c

```
/* SWBus application
 *
 * Application: simpleClient (BASIC EXAMPLE)
 *
 * File: simpleClient.c
 *
 * Author: Hakon Jokstad
 *
 * Date  : 10Mar96 HJo
 *
 */

#include <swbus.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


int main()
{
   SbTSTI localInt;     /* A local integer object */
   SbTSTI theServer;    /* An object representing the server process */
   SbTSTI serverVar;    /* A proxy for a variable in the server */
   SbTSTI serverFunc;   /* A proxy for a function in the server */
   SbTSTI serverMet;    /* A proxy for a method of an object in the server */
   char*  clientName = "simpleClient"; /* The name of the client process */
   char*  serverName = "simpleServer"; /* The name of the server process */
   int    i, j;


   /* Initialize SWBus. Register myself as process "simpleClient".
      Control Server host specified by environment variable CONTROLHOST,
      localhost used if CONTROLHOST is unset */
   if ( SbInit(clientName,NULL,NULL) != SbCOK )
   {
      printf( "SbInit failed for process %s\n",clientName );
      exit( 1 );
   }


   /* Establish connection to a remote process called "simpleServer" */
   printf( "Connecting to remote process %s ...\n", serverName );
   theServer = SbOpen( SbCCRemote, serverName, "~remote", SbCWait );
   printf( "Connection to process %s established\n", SbName(theServer) );


   /* Update a variable in the server */
   serverVar = SbId( theServer, "myInt" );
   i = 49;
   SbPut( serverVar, &i );
   SbFlush( SbCPLocal, theServer );
   printf( "Value %d assigned to variable %s in process %s\n",
           i, SbName(serverVar), SbName(theServer) );
   SbGet( serverVar, &j );
   printf( "Value %d retrieved from variable %s in process %s\n",
           j, SbName(serverVar), SbName(theServer) );
```

```
  /* Call a function in the server */
    serverFunc = SbId( theServer, "myFunc" );
    localInt = SbCreate( SbCPLocal, SbCCInteger, NULL, SbCNull, NULL );
    i = 56;
    SbPut( localInt, &i );
    SbCall( serverFunc, localInt, localInt );
    SbGet( localInt, &j );
    printf( "%s(%d) = %d\n", SbName(serverFunc), i, j );


    /* Call the "terminate" function to terminate the server */
    SbCall( SbId(theServer,"terminate"), SbCNull, SbCNull );


    /* Take down the connection to the server */
    SbDelete( theServer, SbCNull );


    /* Leave the software bus */
    SbExit();


    return 0;
}
```

Appendix C - Advanced Example

System description

The application used in the examples in Chapter 5 simulates a delivery system for a chocolate factory.

The factory can have any number of customers (chocolate shops) and has access to each customer's stock details. Each customer sells the entire line of chocolates that the factory produces.

At regular intervals the factory produces a variety of types of chocolate that are placed in a warehouse. At the end of a production period, the factory examines the stock levels at each of its customers and sends chocolates to those that need new supplies. If the factory does not have enough chocolate to send then the customers will eventually go bankrupt, as they will have nothing to sell.

A simple locking device is used to reduce the risk of the factory updating data in the customer while the customer is also updating its own data. While a factory is distributing chocolates, customers do not sell chocolates, as they are closed for restocking.

If the production rate is higher than the demand then the warehouse will fill up. When the warehouse is almost full, the factory initiates a 'grand chocolate sale' at each customer. During the sale period, the customers sell chocolates at twice the usual rate. The sale is discontinued when the warehouse stocks are sufficiently depleted.

The factory cannot produce more chocolate than there is room for in the warehouse. If there is little production for more than a specified number of time periods, because the stock level in the warehouse is over a specified limit, the factory will go bankrupt.

New customers can connect to the factory at any time, and existing customers can disconnect at will. When restocking customers, those that connected to the factory earliest have priority.

If the factory goes bankrupt or quits, existing customers continue to sell chocolates until their stocks run out. If, however a new factory is started, the customers connect to the new factory and continue trading as before.

A number of assumptions have been made to keep the model simple. They are not important to the understanding of the techniques used and are therefore not described in detail here.

We will now look at how the factory and customer processes have been implemented before explaining how to compile and run the code.

Factory implementation

The factory has the following SWBus objects:

- **`chocolateClass`** (with a name, quantity and capacity)
- **`stockArrayClass`** (an array of chocolateClass objects)
- **`warehouseArray`** (a stockArrayClass object)
- **`customerClass`** (a 'remote' customer class)
- **`customerGroup`** (a group of customerClass objects)
- **`periodic`** (a periodic object)
- **`io`** (an I/O object for handling keyboard input)
- **`lockObject`** (containing a simple Boolean value)

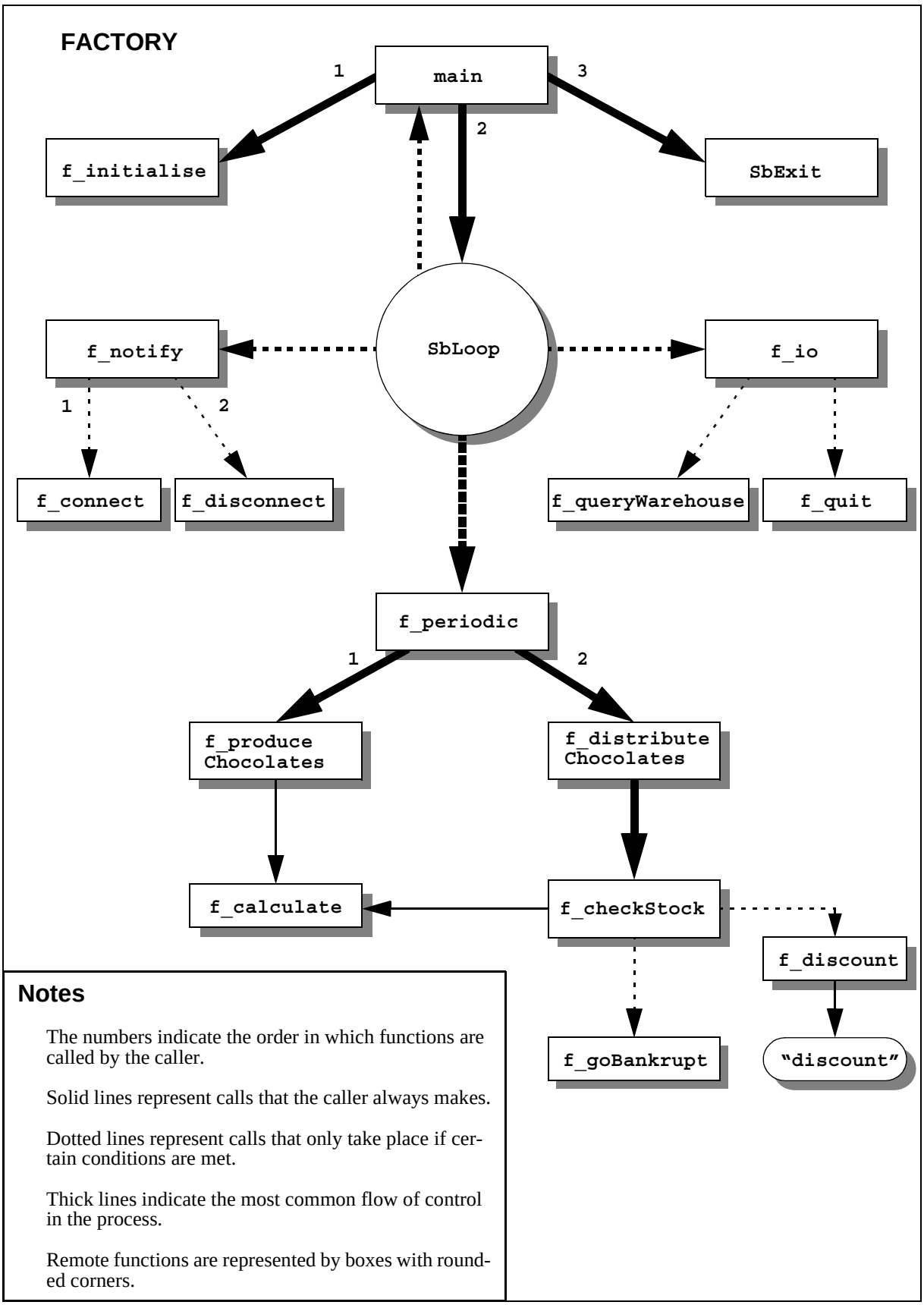
In addition to the SWBus objects, a number of C variables are also used:

- **`timeInterval`** (interval of the periodic object)
- **`productionRate`** (quantity of each chocolate type to produce per period)
- **`lock`** (Boolean value for the lockObject)
- **`discountPeriod`** (Boolean to indicate whether customers are selling chocolate at the 'grand chocolate sale' rate)
- **`crisisCounter`** (number of periods with full warehouse)
- **`crisisLimit`** (if the crisisCounter exceeds the crisisLimit then the factory should go bankrupt)

The periodic and io objects are used to regain control from the SWBus' main loop. The io object is used to query the factory and to issue a quit command. The periodic object calls the function ***f_periodic*** at regular intervals to start a new production and distribution cycle.

When a customer connects, it's proxy is put into the customer group. It is removed from the group when it disconnects.

A chart showing the basic flow of control in the factory process is shown on the next page. Comments about each function can be found at the beginning of each function in the source code itself. Apart from ***SbLoop*** and ***SbExit***, calls to SWBus functions are not shown in the chart. All calls made from ***SbLoop*** are made by the SWBus.


FACTORY
main
1
2
3
f_initialise
SbExit
SbLoop
f_notify
f_io
1
2
f_connect
f_disconnect
f_queryWarehouse
f_quit
f_periodic
1
2
f_produce Chocolates
f_distribute Chocolates
f_calculate
f_checkStock
f_discount
f_goBankrupt
"discount"
Notes
The numbers indicate the order in which functions are called by the caller.
Solid lines represent calls that the caller always makes.
Dotted lines represent calls that only take place if certain conditions are met.
Thick lines indicate the most common flow of control in the process.
Remote functions are represented by boxes with rounded corners.

All of the functions in the factory process return control to their caller when they complete their tasks. Functions such as ***f_quit*** and ***f_goBankrupt*** call ***SbEndLoop***, so, when control is returned to ***SbLoop***, it returns to ***main*** and the process exits.

Note that the actual flow of control is quite complex as some SW-Bus functions take the control temporarily back to the main loop in order to receive data from remote processes and to check for pending io and periodic objects. This means that the values of local SWBus objects can be modified by external processes, and processes can connect/disconnect, in 'parallel' to the 'logical' flow of control. Control is always returned when the current SW-Bus function returns. Care has been taken so that calls to ***f_io*** and ***f_notify*** return control to the main loop quickly, without affecting the main flow of control started by each call to ***f_periodic***.

Customer implementation

The customer process is very similar to the factory process and the comments above about io and periodic objects, and flow of control in the factory process also apply here.

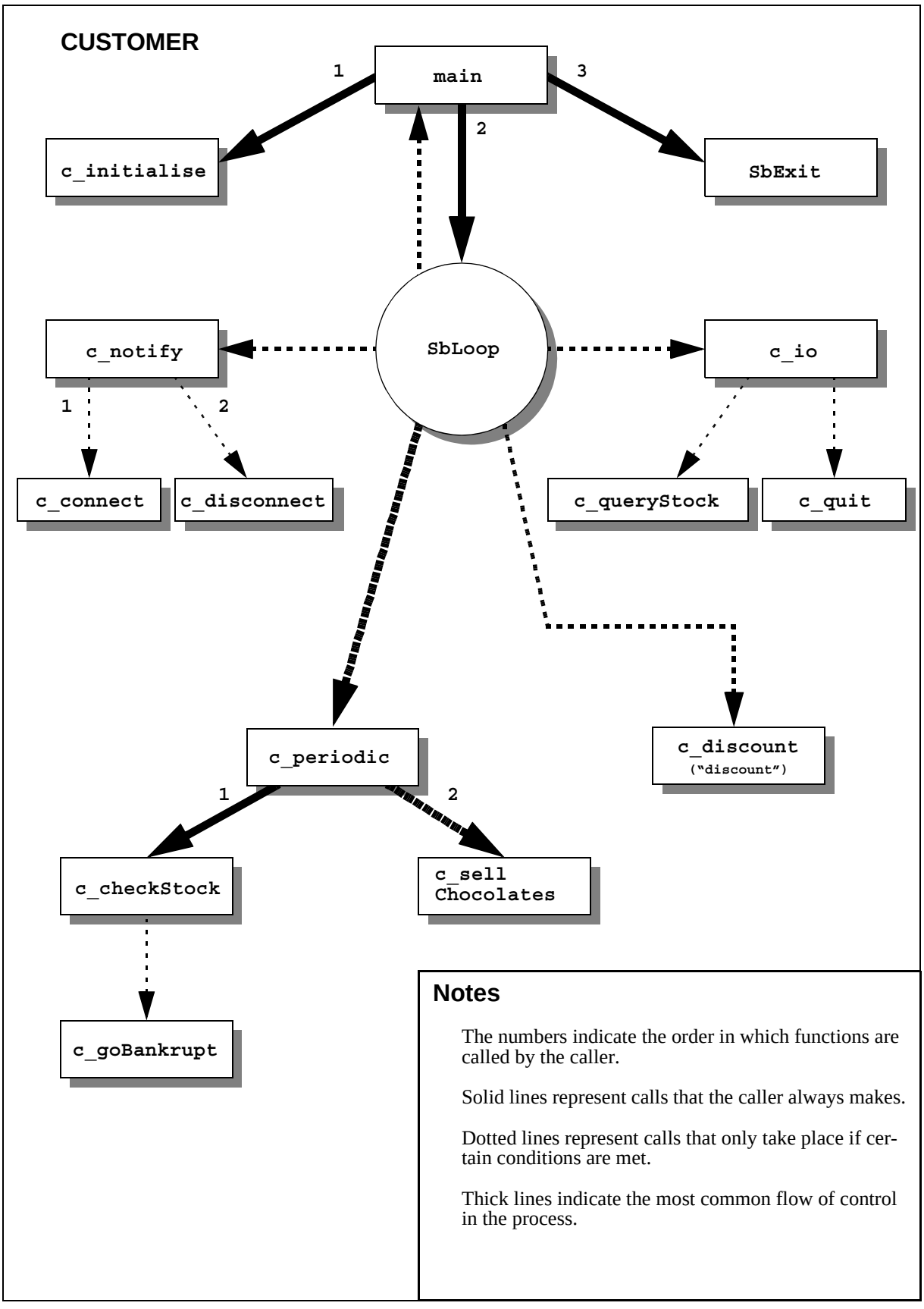
The customer requires the following SWBus objects:

- **`chocolateClass`** (with a name, quantity and capacity)
- **`stockArrayClass`** (an array of chocolateClass objects)
- **`stockArray`** (a stockArrayClass object)
- **`factoryClass`** (a 'remote' factory class)
- **`theFactory`** (the factory process object)
- **`periodic`** (a periodic object)
- **`io`** (an I/O object for handling keyboard input)
- **`lockObject`** (containing a simple Boolean value)

In addition to the SWBus objects, a number of C variables are also needed:

- **`timeInterval`** (interval of the periodic object)
- **`saleRate`** (actual quantity of each chocolate type to sell per period)
- **`normalSaleRate`** (default quantity of each chocolate type to sell per period)
- **`crisisCounter`** (number of periods with no sales)
- **`crisisLimit`** (if the crisisCounter exceeds the crisisLimit then the customer should go bankrupt)

A chart showing the basic flow of control in the customer process is shown on the next page. Comments about each function can be found at the beginning of each function in the source code itself.


CUSTOMER
main
1
2
3
c_initialise
SbExit
SbLoop
c_notify
c_io
1
2
c_connect
c_disconnect
c_queryStock
c_quit
c_periodic
c_discount
("discount")
1
2
c_checkStock
c_sell
Chocolates
c_goBankrupt
Notes
The numbers indicate the order in which functions are called by the caller.
Solid lines represent calls that the caller always makes.
Dotted lines represent calls that only take place if certain conditions are met.
Thick lines indicate the most common flow of control in the process.

Compiling and running the source code

The *README* file at the *examples* directory in the SWBus installation directory contains details about how to compile and link the source code.

Execute each process, **`factory`** and **`customer`**, in the foreground in it's own terminal window. You can start the processes in any order, but ensure that **`control`** is running.

While the processes are running, you can type **`quit`** to exit a process or **`query`** to query it; otherwise each process will run until it 'goes bankrupt' and exits.

Start-up options

You can connect several customer processes to a factory process by starting each customer process with the option **`-n <customer_name>`**, in order to give each process a unique name. The **`-n`** option can be used to specify a user-defined name for the factory when used in conjunction with a factory process, but you must then use the **`-f <factory_name>`** option when starting the customer processes, to specify the name of the factory to connect to. The default names are 'Customer' and 'Factory'.

For example, to start a factory process called 'MyFactory' and two customer processes called 'ShopA' and 'ShopB', type the following lines (one line in each of three terminal windows):

```
% factory  -n MyFactory

% customer -n ShopA -f MyFactory

% customer -n ShopB -f MyFactory
```

You can also run several factory processes, but each customer can only connect to one of the factory processes.

A **`-t`** and a **`-r`** option exists for the customer and the factory processes. The **`-t`** option can be used to specify the time interval in milliseconds between each execution of the periodic function. The **`-r`** option can be used to specify either the sale rate or the production rate, depending on the type of process. The value for the **`-r`** option is a number of chocolates of each chocolate type (the same value is used for each type). The default values are 2000ms and 10 chocolates, for customers, and 5000ms and 100 chocolates, for factories. Enjoy!

factory.c

```
/* SWBus application
 *
 * Application: factory (ADVANCED EXAMPLE)
 *
 * File: factory.c
 *
 * Author: Michael Louka
 *
 * Date  : 15Mar96 MLo
 *
 */


/* Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <swbus.h>

#ifdef sgi
#  include <bstring.h> /* For FD_ZERO */
#endif

/* Definitions */

#define true   1
#define false  0

#define QUANTITY 0
#define CAPACITY 1

/* Global variables */

SbTSTI warehouseArray  = SbCNull;
SbTSTI customerGroup   = SbCNull;

SbTSTI lockObject      = SbCNull;
bool   lock            = false;

int32  timeInterval    = 5000;
int32  productionRate  = 100;

int32  crisisCounter   = 0;
int32  crisisLimit     = 5;

bool   discountPeriod  = false;

const int32 chocolateTypes = 5;
```

```
/* Forward declarations */

bool   f_initialise( char* factoryName );
bool   f_connect( SbTSTI remoteProcessId );
void   f_disconnect( SbTSTI remoteProcessId );
void   f_produceChocolates();
void   f_distributeChocolates();
void   f_checkStock();
int32  f_calculate( int32 option );
void   f_discount( bool startDiscountPeriod );
void   f_queryWarehouse();
void   f_goBankrupt();
void   f_quit();
SbTSTI f_notify( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI f_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI f_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );


bool f_initialise( char* factoryName )
{
   /* Variables for creating the stock array */

   SbTSTI chocolateClass;
   SbTSTI stockArrayClass;
   SbTSTI arrayParameter;
   SbTArrayParameter arrayParameterData;


   /* Variables for setting initial chocolate data */

   SbTSTI currentObjectId;
   char*  chocolateName   = NULL;
   int32  maximumQuantity = 1000;
   int32  startQuantity   = 0;


   /* Variables for defining the remote class */

   SbTSTI customerClass;
   SbTSTI notifyFunction;


   /* Variables for creating the periodic object */

   SbTSTI periodic;
   int32  periodicCount  = -1;
   int32  periodicPeriod = -1;
   SbTSTI periodicIn  = SbCNull;
   SbTSTI periodicOut = SbCNull;
   SbTSTI periodicParameter;
   SbTPeriodicParameter* periodicParameterData;


   /* Variables for creating the keyboard handler
      Keyboard handlers can not be created on win32
      because no socket can be obtained for for stdin */

#ifndef win32
   int32   ioCondition;
   fd_set* ioFdSet;
   SbTSTI  ioIn  = SbCNull;
   SbTSTI  ioOut = SbCNull;
   SbTSTI  io;
   SbTSTI  ioParameter;
   SbTIoParameter* ioParameterData;
#endif /* win32 */
```

```
   /* Variable for the for-loop */

   int32 i;


   /* Initialize the SWBus */

   if ( SbInit( factoryName, NULL, NULL ) != SbCOK )
      return false;


   /* Create the chocolateClass */

   chocolateClass = SbSub( SbCCEmpty,  "ChocolateClass", SbCNull );
   SbAdd( chocolateClass, SbCCString,  "Name",     NULL );
   SbAdd( chocolateClass, SbCCInteger, "Quantity", NULL );
   SbAdd( chocolateClass, SbCCInteger, "Capacity", NULL );


/* Create an array of chocolateClass objects */

   arrayParameterData.baseClass = chocolateClass;
   arrayParameterData.count     = chocolateTypes;

   arrayParameter = SbCreate( SbCPLocal, SbCCArrayParameter, NULL, SbCNull,
                              &arrayParameterData);

   stockArrayClass = SbSub( SbCCArray, "StockArrayClass", arrayParameter );

   SbDelete( arrayParameter, SbCNull );

   warehouseArray = SbCreate( SbCPLocal, stockArrayClass, "WarehouseArray",
                              SbCNull, NULL );


   /* Set initial values of array objects */

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( warehouseArray, i );

      switch ( i )
      {
        case 0:
           chocolateName = strdup( "milk chocolate" );
           break;
        case 1:
           chocolateName = strdup( "dark chocolate" );
           break;
        case 2:
           chocolateName = strdup( "mint wafer" );
           break;
        case 3:
           chocolateName = strdup( "caramel bar" );
           break;
        case 4:
           chocolateName = strdup( "fudge" );
           break;
        default:
           chocolateName = strdup( "unknown" );
      }

      SbPut( SbId( currentObjectId, "Name"     ), &chocolateName   );
      SbPut( SbId( currentObjectId, "Quantity" ), &startQuantity   );
      SbPut( SbId( currentObjectId, "Capacity" ), &maximumQuantity );
   }
```

```
    /* Create subclass of remote and add a user-defined notify function to it */

    notifyFunction = SbDeclare( "FactoryNotify", SbCCType,
                                SbCNull, &f_notify, SbCLC );

    customerClass  = SbSub( SbCCRemote, "CustomerClass", SbCNull );
    SbAdd( customerClass, notifyFunction, "~notify", NULL );


    /* Create a group for the customers to be put into */

    customerGroup = SbCreate( SbCPLocal, SbCCGroup, NULL, SbCNull, NULL );


    /* Create an integer object that will be used as a boolean to
       implement a lock */

    lockObject = SbCreate( SbCPLocal, SbCCInteger, "Lock", SbCNull, &lock );
    /* Create an anonymous periodic object */

    periodicParameter = SbCreate( SbCPLocal, SbCCPeriodicParameter,
                                  NULL, SbCNull, NULL );

    periodicParameterData
       = (SbTPeriodicParameter*)SbData( periodicParameter );

    periodicParameterData->interval = timeInterval;
    periodicParameterData->count    = periodicCount;
    periodicParameterData->period   = periodicPeriod;
    periodicParameterData->term     = NULL;
    periodicParameterData->func     = f_periodic;
    periodicParameterData->inParam  = periodicIn;
    periodicParameterData->outParam = periodicOut;

    periodic = SbCreate( SbCPLocal, SbCCPeriodic,
                         NULL, periodicParameter, NULL );

    SbDelete( periodicParameter, SbCNull );

    /* Create an IO object
       Keyboard handlers can not be created on win32
       because no socket can be obtained for for stdin */

#ifndef win32
    ioCondition = SbCFDREAD;
    ioFdSet     = (fd_set*)malloc( sizeof(fd_set) );

    FD_ZERO( ioFdSet );
    FD_SET( 0, ioFdSet );  /* 0 is socket for stdin */

    ioParameter = SbCreate( SbCPLocal, SbCCIoParameter,
    NULL, SbCNull, NULL );

    ioParameterData = (SbTIoParameter*)SbData( ioParameter );

    ioParameterData->fdc      = ioCondition;
    ioParameterData->fds      = ioFdSet;
    ioParameterData->func     = f_io;
    ioParameterData->inParam  = ioIn;
    ioParameterData->outParam = ioOut;

    io = SbCreate( SbCPLocal, SbCCIo, NULL, ioParameter, NULL );

    SbDelete( ioParameter, SbCNull );
#endif /* win32 */

    return true;
}
```

```
bool f_connect( SbTSTI remoteProcessId )
{
   /* Link the local and remote lock objects and add this customer to
      the customer group */

   SbTSTI remoteLockObject;

   remoteLockObject = SbId( remoteProcessId, "Lock" );

   if ( SbDGood( remoteLockObject ) )
      SbLink( lockObject, remoteLockObject, SbCBPNull );

   SbInclude( customerGroup, SbList( SbCCGroup, remoteProcessId, SbCNull ) );

   printf( "*** Customer \"%s\" connected\n",
   SbName( remoteProcessId ) );

   return true;
}


void f_disconnect( SbTSTI remoteProcessId )
{
   /* Take the remote process out of the customer group */

  SbWithdraw( customerGroup, SbList( SbCCGroup, remoteProcessId, SbCNull ) );

   printf( "*** Customer disconnected.\n" );
}


void f_produceChocolates()
{
   /* Produce one amount of each chocolate type at the production rate,
      but do not produce more than there is room for in the warehouse */

   int32* quantity;
   int32  maximumQuantity;
   SbTSTI currentObjectId;
   int32 i;

   if ( productionRate == 0
        || ( f_calculate( CAPACITY ) - f_calculate( QUANTITY ) == 0 ) )
   {
      printf( "No chocolates produced.\n" );
      return;
   }

   printf( "Producing chocolates...\n" );

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( warehouseArray, i );

      quantity = (int32*)SbData( SbId( currentObjectId, "Quantity" ) );
      SbGet( SbId( currentObjectId, "Capacity" ), &maximumQuantity );

      if ( ( *quantity + productionRate ) > maximumQuantity )
        *quantity = maximumQuantity;
      else
        *quantity = *quantity + productionRate;
   }
}
```

```
void f_distributeChocolates()
{
   /* Check each chocolate type in each customer's stock array and
      send supplies to the customer if they are needed. Do not supply
      more than is available in the warehouse, or more than the customer
     has room to store, and finish by checking the state of the warehouse */

   SbTSTI customerObjectId;
   SbTSTI customerQuantityId;
   SbTSTI customerCapacityId;
   SbTSTI warehouseObjectId;
   SbTSTI warehouseQuantityId;
   int32  customerQuantity;
   int32  customerCapacity;
   int32  warehouseQuantity;
   int32  customerRequires;
   int32  quantitySent;
   int32  newWarehouseQuantity;
   int32  newCustomerQuantity;
   int32  stockLevel = f_calculate( QUANTITY );

   SbTSTI remoteStockArray;
   SbTSTI theCustomer;
   int32 i;
   if ( SbHead( customerGroup ) == SbCNull ) /* empty group? */
   {
      printf( "No customers to distribute to.\n" );
      f_checkStock();
      return;
   }

   lock = true;
   SbSend( lockObject );
   SbFlush( SbCPLocal, customerGroup );

   printf( "Distributing chocolates...\n" );

   theCustomer = SbFirst( customerGroup, SbCNull );

   while ( SbDGood( theCustomer ) && !SbDDead( theCustomer ) )
   {
      remoteStockArray = SbId( theCustomer, "StockArray" );

      if ( SbDGood( remoteStockArray ) )
      {
        for ( i = 0; i < chocolateTypes; i++ )
        {
           customerObjectId  = SbIndex( remoteStockArray, i );
           warehouseObjectId = SbIndex( warehouseArray,   i );

           warehouseQuantityId = SbId( warehouseObjectId, "Quantity" );
           customerQuantityId  = SbId( customerObjectId,  "Quantity" );
           customerCapacityId  = SbId( customerObjectId,  "Capacity" );

           SbGet( warehouseQuantityId, &warehouseQuantity );
           SbGet( customerQuantityId,  &customerQuantity  );
           SbGet( customerCapacityId,  &customerCapacity  );
```

```
            if ( customerCapacity > 0 )
            {
               customerRequires = customerCapacity * 0.5;

               if ( customerQuantity <= customerRequires )
               {
                 quantitySent =
                    ( ( customerCapacity - customerQuantity )
                     <= warehouseQuantity ?
                      customerCapacity - customerQuantity
                      : warehouseQuantity );

                 newWarehouseQuantity = warehouseQuantity - quantitySent;
                 newCustomerQuantity  = customerQuantity  + quantitySent;

                 SbPut( warehouseQuantityId, &newWarehouseQuantity );
                 SbPut( customerQuantityId,  &newCustomerQuantity  );
               }
            }
         }
      }

      theCustomer = SbNext( customerGroup );
   }

   lock = false;
   SbSend( lockObject );
   SbFlush( SbCPLocal, customerGroup );

   f_checkStock();
}


void f_checkStock()
{
   /* If the stock level is greater than 80% of the warehouse capacity
      then initiate a discount period, otherwise stop any ongoing
      discount period. If the warehouse is full then increase the
      crisis counter until the crisis limit is reached, in which case
      go bankrupt */

  int32 stockLevel        = f_calculate( QUANTITY );
  int32 warehouseCapacity = f_calculate( CAPACITY );

  if ( stockLevel >= ( 0.8 * warehouseCapacity ) )
  {
     if ( discountPeriod != true )
       f_discount( true );
  }
  else
     if ( discountPeriod == true )
       f_discount( false );

  if ( stockLevel == warehouseCapacity )
  {
     printf( "No more room in the warehouse!\n" );

     crisisCounter++;
     if ( crisisCounter >= crisisLimit )
       f_goBankrupt();
  }
  else
     printf( "There are %d chocolates in the warehouse.\n", stockLevel );
}
```

```
int32 f_calculate( int32 option )
{
   /* Depending on the input parameter, calculate the total stock
      level or the total capacity of the warehouse */

   int32 quantity;
   int32 count = 0;
   SbTSTI currentObjectId;
   int32 i;

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( warehouseArray, i );
      if ( option == QUANTITY )
        SbGet( SbId( currentObjectId, "Quantity" ), &quantity );
      else if (option == CAPACITY )
        SbGet( SbId( currentObjectId, "Capacity" ), &quantity );
      else
        quantity = 0;
      count += quantity;
   }

   return count;
}



void f_discount( bool startDiscountPeriod )
{
   /* Call the discount function in each customer to initiate a
      discount period */

   SbTSTI discountFunctions;
   SbTSTI inputParameter;

   if ( SbHead( customerGroup ) == SbCNull ) /* empty group? */
      return;

   discountFunctions = SbId( customerGroup, "discount" );

   if ( SbDGood( discountFunctions ) )
   {
      inputParameter = SbCreate( SbCPLocal, SbCCInteger,
                                 NULL, SbCNull, NULL );
      SbPut( inputParameter, &startDiscountPeriod );
      SbCall( discountFunctions, inputParameter, SbCNull );
      SbDelete( inputParameter, SbCNull );
      discountPeriod = startDiscountPeriod;
   }
}



void f_queryWarehouse()
{
   /* Print out the warehouse stock array contents to stdout */

   int32 quantity;
   int32 maximumQuantity;
   char* name;
   SbTSTI currentObjectId;
   int32 i;

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( warehouseArray, i );
      SbGet( SbId( currentObjectId, "Name"     ), &name );
      SbGet( SbId( currentObjectId, "Quantity" ), &quantity );
```

```
      SbGet( SbId( currentObjectId, "Capacity" ), &maximumQuantity );
      printf( "Quantity: %d  Capacity: %d  (%s)\n",
      quantity, maximumQuantity, name );
   }
}


void f_goBankrupt()
{
   /* Issue message and end SWBus loop */

   printf( "\nThe factory has gone bankrupt!\n" );
   SbEndLoop();
}


void f_quit()
{
   /* Issue message and end SWBus loop */

   printf( "\nThe factory has been demolished.\n" );
   SbEndLoop();
}


SbTSTI f_notify( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Call connect if a new customer connects and call
      disconnect if an existing customer disconnects */

   SbTType notification;

   SbGet( in, &notification );

   if ( notification & SbCBPSConnected )
      f_connect( o );

   if ( (notification & SbCBPSDisconnect) || (notification & SbCBPSBroken) )
     f_disconnect( o );

   return out;
}


SbTSTI f_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Produce and distribute chocolate */

   f_produceChocolates();
   f_distributeChocolates();

   return out;
}
```

```
SbTSTI f_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Handle text input from the user */

   char buffer[64];

   scanf( "%s", &buffer );

   if ( strcmp( buffer, "quit" ) == 0 )
      f_quit();
   else
      if ( strcmp( buffer, "query" ) == 0 )
         f_queryWarehouse();

   return out;
}



main( int argc, char* argv[] )
{
   char* factoryName = NULL;
   int   argInd = 1;

   while ( argInd < argc )
   {
      if ( strcmp(argv[argInd++],"-n") == 0 )
         factoryName = strdup( argv[argInd++] );
      else if ( strcmp(argv[argInd++],"-t") == 0 )
         timeInterval = atoi( argv[argInd++] );
      else if ( strcmp(argv[argInd++],"-r") == 0 )
         productionRate = atoi( argv[argInd++] );
      else
      {
         printf( "Unrecognised option.\n" );
         exit(1);
      }
   }

   if ( factoryName == NULL )
      factoryName = strdup( "Factory" );

   printf( "*** Welcome to the Halden Chocolate Factory Simulator ***\n\n" );

   if ( f_initialise( factoryName ) )
   {
      SbLoop();
      SbExit();
   }
   else
      printf( "Initialisation failed. Exiting...\n" );

   if ( factoryName != NULL )
      free( factoryName );

   return 0;
}
```

customer.c

```
/* SWBus application
 *
 * Application: customer (ADVANCED EXAMPLE)
 *
 * File: customer.c
 *
 * Author: Michael Louka
 *
 * Date  : 14Mar96 MLo
 *
 */


/* Includes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <swbus.h>

#ifdef sgi
#  include <bstring.h> /* For FD_ZERO */
#endif

/* Definitions */

#define true   1
#define false  0


/* Global variables */

SbTSTI stockArray     = SbCNull;
SbTSTI theFactory     = SbCNull;
SbTSTI lockObject     = SbCNull;

int32  timeInterval   = 2000;
int32  saleRate       = 10;

int32  crisisCounter  = 0;
int32  crisisLimit    = 5   ;

int32  normalSaleRate = 10; /* saleRate */

const int32 chocolateTypes = 5;


/* Forward function declarations */

bool   c_initialise( char* customerName, char* factoryName );
void   c_connect( SbTSTI remoteProcessId );
void   c_disconnect();
bool   c_checkStock();
void   c_sellChocolates();
void   c_queryStock();
void   c_goBankrupt();
void   c_quit();
SbTSTI c_discount( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI c_notify  ( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI c_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
SbTSTI c_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out );
```

```
bool c_initialise( char* customerName, char* factoryName )
{
   /* Variables for creating the stock array */

   SbTSTI chocolateClass;
   SbTSTI stockArrayClass;
   SbTSTI arrayParameter;
   SbTArrayParameter arrayParameterData;


   /* Variables for setting initial chocolate data */

   SbTSTI currentObjectId;
   char*  chocolateName   = NULL;
   int32  maximumQuantity = 100;
   int32  startQuantity   = 0;


   /* Variables for declaring and creating the discount function */

   SbTSTI functionClass;
   SbTSTI functionObject;


   /* Variables for defining the remote class */

   SbTSTI factoryClass;
   SbTSTI notifyFunction;


   /* Variables for creating the periodic object */

   SbTSTI periodic;
   int32  periodicCount  = -1;
   int32  periodicPeriod = -1;
   SbTSTI periodicIn  = SbCNull;
   SbTSTI periodicOut = SbCNull;
   SbTSTI periodicParameter;
   SbTPeriodicParameter* periodicParameterData;

   /* Variables for creating the keyboard handler
      Keyboard handlers can not be created on win32
      because no socket can be obtained for for stdin */

#ifndef win32
   int32   ioCondition;
   fd_set* ioFdSet;
   SbTSTI  ioIn  = SbCNull;
   SbTSTI  ioOut = SbCNull;
   SbTSTI  io;
   SbTSTI  ioParameter;
   SbTIoParameter* ioParameterData;
#endif /* win32 */

   /* Variable for the for-loop */

   int32 i;

   /* Initialize the SWBus */

   printf( "Initialising...\n" );

   if ( SbInit( customerName, NULL, NULL ) != SbCOK )
      return false;
```

```
/* Create the chocolateClass */

chocolateClass = SbSub( SbCCEmpty, "ChocolateClass", SbCNull );
SbAdd( chocolateClass, SbCCString,  "Name",     NULL );
SbAdd( chocolateClass, SbCCInteger, "Quantity", NULL );
SbAdd( chocolateClass, SbCCInteger, "Capacity", NULL );


/* Create an array of chocolateClass objects */

arrayParameterData.baseClass = chocolateClass;
arrayParameterData.count     = chocolateTypes;

arrayParameter = SbCreate( SbCPLocal, SbCCArrayParameter, NULL, SbCNull,
                           &arrayParameterData);

stockArrayClass = SbSub( SbCCArray, "StockArrayClass", arrayParameter );

SbDelete( arrayParameter, SbCNull );

stockArray = SbCreate( SbCPLocal, stockArrayClass, "StockArray",
                       SbCNull, NULL );


/* Set initial values of array objects */

for ( i = 0; i < chocolateTypes; i++ )
{
   currentObjectId = SbIndex( stockArray, i );

   switch ( i )
   {
     case 0:
       chocolateName = strdup( "milk chocolate" );
       break;
     case 1:
       chocolateName = strdup( "dark chocolate" );
       break;
     case 2:
       chocolateName = strdup( "mint wafer" );
       break;
     case 3:
       chocolateName = strdup( "caramel bar" );
       break;
     case 4:
       chocolateName = strdup( "fudge" );
       break;
    default:
       chocolateName = strdup( "unknown" );
   }

   SbPut( SbId( currentObjectId, "Name"     ), &chocolateName   );
   SbPut( SbId( currentObjectId, "Quantity" ), &startQuantity   );
   SbPut( SbId( currentObjectId, "Capacity" ), &maximumQuantity );
}


/* Create subclass of remote and add a user-defined notify function
   to it, and then open a connection to a factory process */

notifyFunction = SbDeclare( "CustomerNotify", SbCCType,
                            SbCNull, &c_notify, SbCLC );

factoryClass   = SbSub( SbCCRemote, "FactoryClass", SbCNull );
SbAdd ( factoryClass, notifyFunction, "~notify", NULL );

SbOpen( factoryClass, factoryName, "CustomerClass", SbCContinue );
```

```
    /* Declare discount function and creating instance of it */

    functionClass = SbDeclare( "DiscountFunction", SbCCInteger,
                               SbCNull, c_discount, SbCLC );
    functionObject = SbCreate( SbCPLocal, functionClass,
                               "discount", SbCNull, NULL );


    /* Create an integer object that will be a local representation of
       the factory's locking mechanism */

    lockObject = SbCreate( SbCPLocal, SbCCInteger, "Lock", SbCNull, NULL );


    /* Create an anonymous periodic object */

    periodicParameter = SbCreate( SbCPLocal, SbCCPeriodicParameter, NULL,
                                  SbCNull, NULL );

    periodicParameterData
       = (SbTPeriodicParameter*)SbData( periodicParameter );

    periodicParameterData->interval = timeInterval;
    periodicParameterData->count    = periodicCount;
    periodicParameterData->period   = periodicPeriod;
    periodicParameterData->term     = NULL;
    periodicParameterData->func     = c_periodic;
    periodicParameterData->inParam  = periodicIn;
    periodicParameterData->outParam = periodicOut;

    periodic = SbCreate( SbCPLocal, SbCCPeriodic,
                         NULL, periodicParameter, NULL );

    SbDelete( periodicParameter, SbCNull );


    /* Create an IO object
       Keyboard handlers can not be created on win32
       because no socket can be obtained for for stdin */

  #ifndef win32
    ioCondition = SbCFDREAD;
    ioFdSet     = (fd_set*)malloc( sizeof( fd_set ) );

    FD_ZERO( ioFdSet );
    FD_SET( 0, ioFdSet ); /* 0 is socket for stdin */

    ioParameter = SbCreate( SbCPLocal, SbCCIoParameter,
                            NULL, SbCNull, NULL );

    ioParameterData = (SbTIoParameter*)SbData( ioParameter );

    ioParameterData->fdc      = ioCondition;
    ioParameterData->fds      = ioFdSet;
    ioParameterData->func     = c_io;
    ioParameterData->inParam  = ioIn;
    ioParameterData->outParam = ioOut;

    io = SbCreate( SbCPLocal, SbCCIo, NULL, ioParameter, NULL );

    SbDelete( ioParameter, SbCNull );
  #endif /* win32 */

    return true;
  }
```

```
void c_connect( SbTSTI remoteProcessId )
{
   /* Set the factory to the remote process id */

   theFactory = remoteProcessId;

   printf( "*** Factory \"%s\" connected.\n",
   SbName( remoteProcessId ) );
}



void c_disconnect()
{
   /* Reset the stored remote process id,  and revert sale rate
      and the lock value */

   bool lock = false;

   theFactory = SbCNull;

   if ( saleRate != normalSaleRate )
   {
      printf( "Grand sale of chocolate is over (factory disconnected).\n" );
      saleRate = normalSaleRate;
   }

   SbPut( lockObject, &lock );

   printf( "*** Factory disconnected.\n" );
}



bool c_checkStock()
{
   /* Calculate the total number of chocolates in stock,
      increment the crisis counter if out of stock, and
      go bankrupt if the crisis counter has exceeded its
      predefined limit */

   int32  quantity;
   int32  stockLevel = 0;
   SbTSTI currentObjectId;
   int32  i;

   printf( "Checking stock...\n" );

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( stockArray, i );
      SbGet( SbId( currentObjectId, "Quantity" ), &quantity );
      stockLevel += quantity;
   }

   printf( "There are %d chocolates in stock.\n", stockLevel );
```

```
    if ( stockLevel == 0 )
    {
       crisisCounter++;
       if ( crisisCounter == crisisLimit )
            c_goBankrupt();

       return false;
    }
    else
       if ( crisisCounter != 0 )
            crisisCounter = 0;

    return true;
}



void c_sellChocolates()
{
    /* Reduce amount of each chocolate type in stock by saleRate, unless
       not enough in stock or the lock is true, and put changes in SWBus
       send buffer */

    int32  lock = true;
    int32* quantity;
    SbTSTI quantityId;
    SbTSTI currentObjectId;
    int32  i;

    if ( saleRate == 0 )
    {
       printf( "No customers to sell to.\n" );
       return;
    }

    if ( SbDGood( SbGet( lockObject, &lock ) ) )
      if ( lock == true )
      {
        printf( "Shop is closed while restocking...\n" );
        return;
      }

    printf( "Selling chocolates...\n" );

    for ( i = 0; i < chocolateTypes; i++ )
    {
       currentObjectId = SbIndex( stockArray, i );

       quantityId  = SbId( currentObjectId, "Quantity" );
       quantity    = (int32*)SbData( quantityId );

       if ( quantity != NULL)
          if ( *quantity > saleRate )
            *quantity = *quantity - saleRate;
          else
            *quantity = 0;
    }
}



void c_queryStock()
{
    /* Print the stock array contents to stdout */

    int32  quantity;
    int32  maximumQuantity;
    char*  name;
```

```
   SbTSTI currentObjectId;
   int32  i;

   for ( i = 0; i < chocolateTypes; i++ )
   {
      currentObjectId = SbIndex( stockArray, i );
      SbGet( SbId( currentObjectId, "Name"     ), &name );
      SbGet( SbId( currentObjectId, "Quantity" ), &quantity );
      SbGet( SbId( currentObjectId, "Capacity" ), &maximumQuantity );
      printf( "Quantity: %d  Capacity: %d  (%s)\n", quantity,
      maximumQuantity, name );
   }
}


void c_goBankrupt()
{
   /* Issue message and end SWBus loop */

   printf( "\nThe shop has gone bankrupt!\n" );
   SbEndLoop();
}


void c_quit()
{
   /* Issue message and end SWBus loop */

   printf( "\nThe shop has been demolished.\n" );
   SbEndLoop();
}


SbTSTI c_discount( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Extract the input parameter, double normal sale rate if true or
      reset sale rate to the normal rate if false */

   bool discountPeriod;

   SbGet( in, &discountPeriod );

   if ( discountPeriod == true )
   {
      saleRate = 4 * normalSaleRate;
      printf( "Grand sale of chocolate started!\n" );
   }
   else
      if ( saleRate != normalSaleRate )
      {
         saleRate = normalSaleRate;
         printf( "Grand sale of chocolate ended.\n" );
      }

   return out;
}
```

```
SbTSTI c_notify( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Call connect if a process connects and call disconnect
      if the factory disconnects */

   SbTType notification;

   SbGet( in, &notification );

   if ( notification & SbCBPSConnected )
      c_connect( o );

   if ( (notification & SbCBPSDisconnect) || (notification & SbCBPSBroken) )
     c_disconnect();

   return out;
}



SbTSTI c_periodic( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Check stock and sell chocolates (if any). Also check the crisis
      counter to make sure that the shop is not bankrupt */

   if ( c_checkStock() )
      c_sellChocolates();
   else
      if ( crisisCounter >= crisisLimit )
 printf( "No chocolates to sell.\n" );

   return out;
}



SbTSTI c_io( SbTSTI o, SbTSTI f, SbTSTI in, SbTSTI out )
{
   /* Handle text input from the user */

   char buffer[64];

   scanf( "%s", &buffer );

   if ( strcmp( buffer, "quit" ) == 0 )
      c_quit();
   else
      if ( strcmp( buffer, "query" ) == 0 )
        c_queryStock();

   return out;
}
```

```
main( int argc, char* argv[] )
{
   char* customerName = NULL;
   char* factoryName  = NULL;
   int   argInd = 1;

   while ( argInd < argc )
   {
      if ( strcmp(argv[argInd++],"-n") == 0 )
         customerName = strdup( argv[argInd++] );
      if ( strcmp(argv[argInd++],"-f") == 0 )
         factoryName = strdup( argv[argInd++] );
      else if ( strcmp(argv[argInd++],"-t") == 0 )
         timeInterval = atoi( argv[argInd++] );
      else if ( strcmp(argv[argInd++],"-r") == 0 )
      {
         saleRate = atoi( argv[argInd++] );
         normalSaleRate = saleRate;
      }
      else
      {
         printf( "Unrecognised option.\n" );
         exit(1);
      }
   }

   if ( customerName == NULL )
      customerName = strdup( "Customer" );

   if ( factoryName == NULL )
      factoryName  = strdup( "Factory" );

   printf( "*** Welcome to the Halden Chocolate Shop Simulator ***\n\n" );

   if ( c_initialise( customerName, factoryName ) )
   {
      SbLoop();
      SbExit();
   }
   else
      printf( "Initialisation failed. Exiting...\n" );

   if ( customerName != NULL )
      free( customerName );

   if ( factoryName != NULL )
      free( factoryName );

   return 0;
}
```